

Hands-on Algorithmic Problem Solving

Data Structures, Algorithms, Python
Modules and Coding Interview Problem
Patterns

LI YIN¹

February 6, 2022

¹<https://liyinscience.com>

Contents

0 Preface	1
1 Reading of This Book	7
1.1 Structures	7
1.2 Reading Suggestions	10
I Introduction	13
2 The Global Picture of Algorithmic Problem Solving	15
2.1 Introduction	15
2.1.1 What?	16
2.1.2 How?	17
2.1.3 Organization of the Contents	18
2.2 Introduction	18
2.3 Problem Modeling	19
2.3.1 Understand Problems	20
2.3.2 Understand Solution Space	22
2.4 Problem Solving	25
2.4.1 Apply Design Principle	25
2.4.2 Algorithm Design and Analysis Principles	26
2.4.3 Algorithm Categorization	28
2.5 Programming Languages	28
2.6 Tips for Algorithm Design	29
2.7 Exercise	29
2.7.1 Knowledge Check	29

3 Coding Interviews and Resources	33
3.1 Tech Interviews	33
3.1.1 Coding Interviews and Hiring Process	33
3.1.2 Why Coding Interviews?	35
3.2 Tips and Resources	36
3.2.1 Tips	36
3.2.2 Resources	38
II Warm Up: Abstract Data Structures and Tools	43
4 Abstract Data Structures	47
4.1 Introduction	47
4.2 Linear Data Structures	48
4.2.1 Array	48
4.2.2 Linked List	49
4.2.3 Stack and Queue	50
4.2.4 Hash Table	51
4.3 Graphs	56
4.3.1 Introduction	56
4.3.2 Types of Graphs	56
4.3.3 Reference	59
4.4 Trees	59
4.4.1 Introduction	59
4.4.2 N-ary Trees and Binary Tree	62
5 Introduction to Combinatorics	65
5.1 Permutation	66
5.1.1 n Things in m positions	66
5.1.2 Recurrence Relation and Math Induction	67
5.1.3 See Permutation in Problems	67
5.2 Combination	67
5.2.1 Recurrence Relation and Math Induction	68
5.3 Partition	68
5.3.1 Integer Partition	69
5.3.2 Set Partition	69
5.4 Array Partition	71
5.5 Merge	72
5.6 More Combinatorics	72
6 Recurrence Relations	75
6.1 Introduction	75
6.2 General Methods to Solve Linear Recurrence Relation	78
6.2.1 Iterative Method	78

6.2.2	Recursion Tree	79
6.2.3	Mathematical Induction	80
6.3	Solve Homogeneous Linear Recurrence Relation	81
6.4	Solve Non-homogeneous Linear Recurrence Relation	83
6.5	Useful Math Formulas	84
6.6	Exercises	84
6.7	Summary	84
III	Get Started: Programming and Python Data Structures	85
7	Iteration and Recursion	89
7.1	Introduction	89
7.2	Iteration	90
7.3	Factorial Sequence	91
7.4	Recursion	92
7.5	Iteration VS Recursion	94
7.6	Exercises	96
7.7	Summary	96
8	Bit Manipulation	97
8.1	Python Bitwise Operators	97
8.2	Python Built-in Functions	99
8.3	Twos-complement Binary	100
8.4	Useful Combined Bit Operations	102
8.5	Applications	104
8.6	Exercises	109
9	Python Data Structures	111
9.1	Introduction	111
9.2	Array and Python Sequence	112
9.2.1	Introduction to Python Sequence	112
9.2.2	Range	113
9.2.3	String	114
9.2.4	List	116
9.2.5	Tuple	122
9.2.6	Summary	124
9.2.7	Bonus	124
9.2.8	Exercises	124
9.3	Linked List	126
9.3.1	Singly Linked List	126
9.3.2	Doubly Linked List	130
9.3.3	Bonus	132

9.3.4	Hands-on Examples	132
9.3.5	Exercises	134
9.4	Stack and Queue	134
9.4.1	Basic Implementation	135
9.4.2	Deque: Double-Ended Queue	137
9.4.3	Python built-in Module: Queue	138
9.4.4	Bonus	140
9.4.5	Exercises	140
9.5	Hash Table	142
9.5.1	Implementation	142
9.5.2	Python Built-in Data Structures	145
9.5.3	Exercises	148
9.6	Graph Representations	149
9.6.1	Introduction	149
9.6.2	Use Dictionary	152
9.7	Tree Data Structures	153
9.7.1	LeetCode Problems	155
9.8	Heap	157
9.8.1	Basic Implementation	158
9.8.2	Python Built-in Library: <code>heapq</code>	162
9.9	Priority Queue	166
9.10	Bonus	171
9.11	Exercises	171

IV Core Principle: Algorithm Design and Analysis 173

10 Algorithm Complexity Analysis	177	
10.1	Introduction	178
10.2	Asymptotic Notations	180
10.3	Practical Guideline	183
10.4	Time Recurrence Relation	184
10.4.1	General Methods to Solve Recurrence Relation	185
10.4.2	Solve Divide-and-Conquer Recurrence Relations	188
10.4.3	Hands-on Example: Insertion Sort	190
10.5	*Amortized Analysis	192
10.6	Space Complexity	192
10.7	Summary	193
10.8	Exercises	194
10.8.1	Knowledge Check	194

11 Search Strategies	195
11.1 Introduction	195
11.2 Uninformed Search Strategies	198
11.2.1 Breath-first Search	199
11.2.2 Depth-first Search	201
11.2.3 Uniform-Cost Search(UCS)	203
11.2.4 Iterative-Deepening Search	204
11.2.5 Bidirectional Search**	206
11.2.6 Summary	208
11.3 Graph Search	208
11.3.1 Depth-first Search in Graph	210
11.3.2 Breadth-first Search in Graph	215
11.3.3 Depth-first Graph Search	218
11.3.4 Breadth-first Graph Search	222
11.4 Tree Traversal	224
11.4.1 Depth-First Tree Traversal	224
11.4.2 Iterative Tree Traversal	227
11.4.3 Breadth-first Tree Traversal	231
11.5 Informed Search Strategies**	232
11.5.1 Best-first Search	232
11.5.2 Hands-on Examples	233
11.6 Exercises	234
11.6.1 Coding Practice	234
12 Combinatorial Search	235
12.1 Introduction	235
12.2 Backtracking	238
12.2.1 Introduction	238
12.2.2 Permutations	239
12.2.3 Combinations	245
12.2.4 More Combinatorics	247
12.2.5 Backtracking in Action	250
12.3 Solving CSPs	251
12.4 Solving Combinatorial Optimization Problems	254
12.4.1 Knapsack Problem	256
12.4.2 Travelling Salesman Problem	260
12.5 Exercises	261
13 Reduce and Conquer	263
13.1 Introduction	263
13.2 Divide and Conquer	265
13.2.1 Concepts	265
13.2.2 Hands-on Examples	267
13.3 Constant Reduction	269

13.3.1 Concepts	269
13.3.2 Hands-on Examples	270
13.4 Divide-and-conquer VS Constant Reduction	271
13.5 A to B	272
13.5.1 Concepts	272
13.5.2 Practical Guideline and Examples	272
13.6 The Skyline Problem	273
13.7 Exercises	273
14 Decrease and Conquer	275
14.1 Introduction	275
14.2 Binary Search	276
14.2.1 Lower Bound and Upper Bound	277
14.2.2 Applications	281
14.3 Binary Search Tree	285
14.3.1 Operations	286
14.3.2 Binary Search Tree with Duplicates	293
14.4 Segment Tree	294
14.4.1 Implementation	296
14.5 Exercises	299
14.5.1 Exercises	300
15 Sorting and Selection Algorithms	303
15.1 Introduction	303
15.2 Python Comparison Operators and Built-in Functions	305
15.3 Naive Sorting	308
15.3.1 Insertion Sort	308
15.3.2 Bubble Sort and Selection Sort	310
15.4 Asymptotically Best Sorting	313
15.4.1 Merge Sort	314
15.4.2 HeapSort	316
15.4.3 Quick Sort and Quick Select	316
15.5 Linear Sorting	320
15.5.1 Bucket Sort	320
15.5.2 Counting Sort	322
15.5.3 Radix Sort	326
15.6 Python Built-in Sort	331
15.7 Summary and Bonus	334
15.8 LeetCode Problems	335
16 Dynamic Programming	341
16.1 Introduction to Dynamic Programming	343
16.1.1 Concepts	343
16.1.2 From Complete Search to Dynamic Programming	345

16.1.3 Fibonacci Sequence	346
16.2 Dynamic Programming Knowledge Base	348
16.2.1 When? Two properties	348
16.2.2 How? Five Elements and Steps	350
16.2.3 Which? Tabulation or Memoization	352
16.3 Hands-on Examples (Main-course Examples)	352
16.3.1 Exponential Problem: Triangle	353
16.3.2 Polynomial Problem: Maximum Subarray	355
16.4 Exercises	357
16.4.1 Knowledge Check	357
16.4.2 Coding Practice	357
16.5 Summary	371
17 Greedy Algorithms	375
17.1 Exploring	376
17.2 Introduction to Greedy Algorithm	380
17.3 *Proof	383
17.3.1 Introduction	383
17.3.2 Greedy Stays Ahead	385
17.3.3 Exchange Arguments	386
17.4 Design Greedy Algorithm	390
17.5 Classical Problems	391
17.5.1 Scheduling	391
17.5.2 Partition	398
17.5.3 Data Compression, File Merge	400
17.5.4 Fractional S	401
17.5.5 Graph Algorithms	401
17.6 Exercises	401
18 Hands-on Algorithmic Problem Solving	403
18.1 Direct Approach	403
18.1.1 Search in Graph	403
18.1.2 Self-Reduction	404
18.1.3 Dynamic Programming	405
18.2 A to B	405
18.2.1 Self-Reduction	405
18.2.2 Dynamic Programming	406
18.2.3 Divide and Conquer	407
V Classical Algorithms	411
19 Advanced Search on Linear Data Structures	415
19.1 Slow-Faster Pointers	416

19.1.1	Array	416
19.1.2	Minimum Window Substring (L76, hard)	419
19.1.3	When Two Pointers do not work	421
19.1.4	Linked List	421
19.2	Opposite-directional Pointers	426
19.3	Follow Up: Three Pointers	427
19.4	Summary	429
19.5	Exercises	430
20	Advanced Graph Algorithms	431
20.1	Cycle Detection	432
20.2	Topological Sort	434
20.3	Connected Components	438
20.3.1	Connected Components Detection	439
20.3.2	Strongly Connected Components	442
20.4	Minimum Spanning Trees	444
20.4.1	Kruskal's Algorithm	445
20.4.2	Prim's Algorithm	448
20.5	Shortest-Paths Algorithms	453
20.5.1	Algorithm Design	454
20.5.2	The Bellman-Ford Algorithm	461
20.5.3	Dijkstra's Algorithm	467
20.5.4	All-Pairs Shortest Paths	469
21	Advanced Data Structures	475
21.1	Monotone Stack	475
21.2	Disjoint Set	480
21.2.1	Basic Implementation with Linked-list or List	481
21.2.2	Implementation with Disjoint-set Forests	483
21.3	Fibonacci Heap	489
21.4	Exercises	489
21.4.1	Knowledge Check	489
21.4.2	Coding Practice	489
22	String Pattern Matching Algorithms	491
22.1	Exact Single-Pattern Matching	491
22.1.1	Prefix Function and Knuth Morris Pratt (KMP)	493
22.1.2	More Applications of Prefix Functions	499
22.1.3	Z-function	499
22.2	Exact Multi-Patterns Matching	503
22.2.1	Suffix Trie/Tree/Array Introduction	503
22.2.2	Suffix Array and Pattern Matching	503
22.2.3	Rabin-Karp Algorithm (Exact or anagram Pattern Matching)	509

22.3 Bonus	509
22.4 Trie for String	510
VI Math and Geometry	517
23 Math and Probability Problems	519
23.1 Numbers	519
23.1.1 Prime Numbers	519
23.1.2 Ugly Numbers	521
23.1.3 Combinatorics	523
23.2 Intersection of Numbers	526
23.2.1 Greatest Common Divisor	526
23.2.2 Lowest Common Multiple	527
23.3 Arithmetic Operations	528
23.4 Probability Theory	529
23.5 Linear Algebra	530
23.6 Geometry	530
23.7 Miscellaneous Categories	532
23.7.1 Floyd's Cycle-Finding Algorithm	532
23.8 Exercise	533
23.8.1 Number	533
VII Problem-Patterns	535
24 Array Questions(15%)	537
24.1 Subarray	538
24.1.1 Absolute-conditioned Subarray	540
24.1.2 Vague-conditioned subarray	547
24.1.3 LeetCode Problems and Misc	552
24.2 Subsequence (Medium or Hard)	556
24.2.1 Others	558
24.3 Subset(Combination and Permutation)	560
24.3.1 Combination	561
24.3.2 Combination Sum	564
24.3.3 K Sum	568
24.3.4 Permutation	573
24.4 Merge and Partition	574
24.4.1 Merge Lists	574
24.4.2 Partition Lists	574
24.5 Intervals	574
24.5.1 Speedup with Sweep Line	576
24.5.2 LeetCode Problems	578

24.6 Intersection	579
24.7 Miscellaneous Questions	580
24.8 Exercises	581
24.8.1 Subsequence with (DP)	581
24.8.2 Subset	586
24.8.3 Intersection	587
25 Linked List, Stack, Queue, and Heap Questions (12%)	589
25.1 Linked List	589
25.2 Queue and Stack	591
25.2.1 Implementing Queue and Stack	591
25.2.2 Solving Problems Using Queue	592
25.2.3 Solving Problems with Stack and Monotone Stack	593
25.3 Heap and Priority Queue	601
26 String Questions (15%)	605
26.1 Ad Hoc Single String Problems	606
26.2 String Expression	606
26.3 Advanced Single String	606
26.3.1 Palindrome	606
26.3.2 Calculator	612
26.3.3 Others	616
26.4 Exact Matching: Sliding Window and KMP	616
26.5 Anagram Matching: Sliding Window	616
26.6 Exact Matching	617
26.6.1 Longest Common Subsequence	617
26.7 Exercise	617
26.7.1 Palindrome	617
27 Tree Questions(10%)	619
27.1 Binary Search Tree	619
27.2 Segment Tree	626
27.3 Trie for String	630
27.4 Bonus	636
27.5 LeetCode Problems	637
28 Graph Questions (15%)	639
28.1 Basic BFS and DFS	639
28.1.1 Explicit BFS/DFS	639
28.1.2 Implicit BFS/DFS	639
28.2 Connected Components	641
28.3 Islands and Bridges	645
28.4 NP-hard Problems	647

29 Dynamic Programming Questions (15%)	651
29.1 Single Sequence $O(n)$	652
29.1.1 Easy Type	653
29.1.2 Subarray Sum: Prefix Sum and Kadane's Algorithm	657
29.1.3 Subarray or Substring	662
29.1.4 Exercise	664
29.2 Single Sequence $O(n^2)$	664
29.2.1 Subsequence	664
29.2.2 Splitting	665
29.3 Single Sequence $O(n^3)$	671
29.3.1 Interval	671
29.4 Coordinate: BFS and DP	677
29.4.1 One Time Traversal	677
29.4.2 Multiple-time Traversal	683
29.4.3 Generalization	689
29.5 Double Sequence: Pattern Matching DP	690
29.5.1 Longest Common Subsequence	691
29.5.2 Other Problems	692
29.5.3 Summary	698
29.6 Knapsack	698
29.6.1 0-1 Knapsack	699
29.6.2 Unbounded Knapsack	701
29.6.3 Bounded Knapsack	702
29.6.4 Generalization	702
29.6.5 LeetCode Problems	703
29.7 Exercise	705
29.7.1 Single Sequence	705
29.7.2 Coordinate	705
29.7.3 Double Sequence	709
VIII Appendix	711
30 Cool Python Guide	713
30.1 Python Overview	714
30.1.1 Understanding Objects and Operations	714
30.1.2 Python Components	717
30.2 Data Types and Operators	719
30.2.1 Arithmetic Operators	720
30.2.2 Assignment Operators	720
30.2.3 Comparison Operators	720
30.2.4 Logical Operators	720
30.2.5 Special Operators	721
30.3 Function	722

30.3.1	Python Built-in Functions	722
30.3.2	Lambda Function	722
30.3.3	Map, Filter and Reduce	723
30.4	Class	725
30.4.1	Special Methods	725
30.4.2	Class Syntax	726
30.4.3	Nested Class	726
30.5	Shallow Copy and the deep copy	727
30.5.1	Shallow Copy using Slice Operator	727
30.5.2	Iterables, Generators, and Yield	728
30.5.3	Deep Copy using copy Module	728
30.6	Global Vs nonlocal	730
30.7	Loops	730
30.8	Special Skills	730
30.9	Supplemental Python Tools	731
30.9.1	Re	731
30.9.2	Bitsect	731
30.9.3	collections	732

List of Figures

1.1	Four umbrellas: each row indicates corresponding parts as outlined in this book.	8
2.1	The State Space Graph. This may appears as a tree, but we can redraw it as a graph.	23
2.2	State Transfer process on a linear structure	24
2.3	State Transfer Process on the tree	25
2.4	Linear Search on explicit linear data structure	25
2.5	Binary Search on an implicit Tree Structure	26
2.6	The State Spaces Graph	30
2.7	State Transfer Tree Structure for LIS, each path represents a possible solution. Each arrow represents an move: find an element in the following elements that's larger than the current node.	31
3.1	Computer Prices, Computer Speed and Cost/MHz	35
3.2	Topic tags on LeetCode	39
3.3	Use Test Case to debug	39
3.4	Use Test Case to debug	40
4.1	Array Representation	48
4.2	Singly Linked List	50
4.3	Doubly Linked List	50
4.4	Stack VS Queue	51
4.5	Example of Hashing Table, replace key as index	52
4.6	Hashtable chaining to resolve the collision, change it to the real example	54

4.7 Example of graphs. Middle: undirected graph, Right: directed graph, and Left: representing undirected graph as directed, Rightmost: weighted graph.	56
4.8 Bipartite Graph	58
4.9 Example of Trees. Left: Free Tree, Right: Rooted Tree with height and depth denoted	60
4.10 A 6-ary Tree Vs a binary tree.	62
4.11 Example of different types of binary trees	63
 6.1 The process to construct a recursive tree for $T(n) = 3T(\lfloor n/4 \rfloor) + O(n)$. There are totally k+1 levels. Use a better figure.	79
 7.1 Iteration vs recursion: in recursion, the line denotes the top-down process and the dashed line is the bottom-up process.	89
7.2 Call stack of recursion function	92
 8.1 Two's Complement Binary for Eight-bit Signed Integers.	100
 9.1 Linked List Structure	126
9.2 Doubly Linked List	130
9.3 Four ways of graph representation, reenumerate it from 0. Redraw the graph	149
9.4 Max-heap be visualized with binary tree structure on the left, and be implemented with Array on the right.	157
9.5 A Min-heap.	158
9.6 Left: delete node 5, and move node 12 to root. Right: 6 is the smallest among 12, 6, and 7, swap node 6 with node 12.	160
9.7 Heapify: The last parent node 45.	162
9.8 Heapify: On node 1	162
9.9 Heapify: On node 21.	162
 10.1 Order of Growth of Common Functions	181
10.2 Graphical examples for asymptotic notations. Replace $f(n)$ with $T(n)$	182
10.3 The process to construct a recursive tree for $T(n) = 3T(\lfloor n/4 \rfloor) + O(n)$. There are totally k+1 levels. Use a better figure.	186
10.4 The cheat sheet for time and space complexity with recurrence function. If $T(n) = T(n-1)+T(n-2)+\dots+T(1)+O(n-1) = 3^n$. They are called factorial, exponential, quadratic, linearithmic, linear, logarithmic, constant.	193
 11.1 Graph Searching	197
11.2 Exemplary Acyclic Graph.	199

11.3	Breath-first search on a simple search tree. At each stage, the node to be expanded next is indicated by a marker.	200
11.4	Depth-first search on a simple search tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory as node L disappears. Dark gray marks nodes that is being explored but not finished.	201
11.5	Bidirectional search.	206
11.6	Exemplary Graph: Free Tree, Directed Cyclic Graph, and Undirected Cyclic Graph.	209
11.7	Search Tree for Exemplary Graph: Free Tree and Directed Cyclic Graph, and Undirected Cyclic Graph.	212
11.8	Depth-first Graph Search Tree.	214
11.9	Breath-first Graph Search Tree.	217
11.10	The process of Depth-first Graph Search in Directed Graph. The black arrows denotes the the relation of u and its not visited neighbors v . And the red arrow marks the backtrack edge.	219
11.11	Classification of Edges: black marks tree edge, red marks back edge, yellow marks forward edge, and blue marks cross edge.	220
11.12	The process of Breath-first Graph Search. The black arrows denotes the the relation of u and its not visited neighbors v . And the red arrow marks the backtrack edge.	223
11.13	Exemplary Binary Tree	224
11.14	Left: PreOrder, Middle: InOrder, Right: PostOrder. The red arrows marks the traversal ordering of nodes.	225
11.15	The process of iterative preorder tree traversal.	227
11.16	The process of iterative postorder tree traversal.	228
11.17	The process of iterative tree traversal.	230
11.18	Draw the breath-first traversal order	231
12.1	A Sudoku puzzle and its solution	236
12.2	The search tree of permutation	240
12.3	The search tree of permutation by swapping. The indexes of items to be swapped are represented as a two element tuple.	241
12.4	The search tree of permutation with repetition	242
12.5	The Search Tree of Combination.	245
12.6	Acyclic graph	248
12.7	The Search Tree of subsequences.The red circled nodes are redundant nodes. Each node has a variable s to indicate the starting index of candidates to add to current subsequence. i indicate the candidate to add to the current node.	250
12.8	A Sudoku puzzle and its solution	252

12.9 Depth-First Branch and bound	258
12.10A complete undirected weighted graph.	260
13.1 Divide and Conquer Diagram	265
13.2 Merge Sort with non-overlapping subproblems where subproblems form a tree	268
13.3 Fibonacci number with overlapping subproblems where subproblems form a graph.	270
14.1 Example of Binary Search	276
14.2 Binary Search: Lower Bound of target 4.	278
14.3 Binary Search: Upper Bound of target 4.	278
14.4 Binary Search: Lower and Upper Bound of target 5 is the same.	278
14.5 Example of Binary search tree of depth 3 and 8 nodes.	285
14.6 The red colored path from the root down to the position where the key 9 is inserted. The dashed line indicates the link in the tree that is added to insert the item.	287
14.7 A BST with nodes 3 duplicated twice.	294
14.8 A BST with nodes 3 marked with two occurrence.	294
14.9 A Segment Tree	295
14.10Illustration of Segment Tree for Sum Range Query.	297
15.1 The whole process for insertion sort: Gray marks the item to be processed, and yellow marks the position after which the gray item is to be inserted into the sorted region.	309
15.2 One pass for bubble sort	311
15.3 The whole process for Selection sort	312
15.4 Merge Sort: The dividing process is marked with dark arrows and the merging process is with gray arrows with the merge list marked in gray color too.	315
15.5 Lomuto's Partition. Yellow, while, and gray marks as region (1), (2) and (3), respectively.	318
15.6 Bucket Sort	321
15.7 Counting Sort: The process of counting occurrence and compute the prefix sum.	323
15.8 Counting sort: Sort keys according to prefix sum.	324
15.9 Radix Sort: LSD sorting integers in iteration	327
15.10Radix Sort: MSD sorting strings in recursion. The black and grey arrows indicate the forward and backward pass in recursion, respectively.	329
15.11The time complexity for common sorting algorithms	335
16.1 Dynamic Programming Chapter Recap	341
16.2 Subproblem Graph	343
16.3 State Transfer for the panlindrom splitting	370

16.4 Summary of different type of dynamic programming problems	373
17.1 All intervals sorted by start and end time.	377
17.2 All intervals	392
17.3 All intervals sorted by start and end time.	393
17.4 Left: sort by start time, Right: sort by finish time	397
17.5 Left: sort by start time, Right: sort by finish time	397
18.1 Graph Model for LIS, each path represents a possible solution.	406
18.2 The solution to LIS.	407
19.1 Two pointer Technique	415
19.2 The data structures to track the state of window.	419
19.3 The partial process of applying two pointers. The grey shaded arrow indicates the pointer that is on move.	420
19.4 Slow-fast pointer to find middle	422
19.5 Circular Linked List	423
19.6 Floyd's Cycle finding Algorithm	424
19.7 Sliding Window Property	429
20.1 Undirected Cyclic Graph. $(0, 1, 2, 0)$ is a cycle	432
20.2 Directed Cyclic Graph, $(0, 1, 2, 0)$ is a cycle.	432
20.3 DAG 1	435
20.4 The connected components in undirected graph, each dashed red circle marks a connected component.	438
20.5 The strongly connected components in directed graph, each dashed red circle marks a strongly connected component.	438
20.6 A graph with four SCCs.	442
20.7 Example of minimum spanning tree in undirected graph, the green edges are edges of the tree, and the yellow filled vertices are vertices of MST (change this to a graph with multiple spanning tree, and highlight the one with the minimum ones).	445
20.8 The process of Kruskal's Algorithm	446
20.9 A cut denoted with red curve partition V into $\{1,2,3\}$ and $\{4,5\}$.	448
20.10 Prim's Algorithm, at each step, we manage the cross edges.	449
20.11 Prim's Algorithm	451
20.12 A weighted and directed graph.	454
20.13 All paths from source vertex s for graph in Fig. 20.12 and its shortest paths.	456
20.14 The simple graph and its adjacency matrix representation (changing it to lower letter)	457
20.15 DP process using Eq. 20.4 for Fig. 20.14	458
20.16 DP process using Eq. 20.5 for Fig. 20.14	459
20.17 DP process using Eq. 20.6 for Fig. 20.14	460

20.18	The update on D for Fig. 20.12. The gray filled spot marks the nodes that updated its estimate value, with its predecessor indicated by incoming red arrow.	462
20.19	The tree structure indicates the updates on D , and the shortest path tree marked by red arrows.	463
20.20	The execution of Bellman-Ford's Algorithm with ordering $[s, t, y, z, x]$	465
20.21	The execution of Bellman-Ford's Algorithm on DAG using topologically sorted vertices. The red color marks the shortest-paths tree.	466
20.22	The execution of Dijkstra's Algorithm on non-negative weighted graph. Red circled vertices represent the priority queue, and blue circled vertices represent the set S . Eventually, the blue colored edges represent the shortest-paths tree.	468
20.23	All shortest-path trees starting from each vertex.	472
21.1	The process of decreasing monotone stack	476
21.2	The connected components using disjoint set.	482
21.3	A disjoint forest	484
22.1	The process of the brute force exact pattern matching	492
22.2	The Skipping Rule	494
22.3	The Sliding Rule	495
22.4	Proof of Lemma	495
22.5	Z function property	500
22.6	Cyclic Shifts	504
22.7	Building a Trie from Patterns	509
22.8	Trie VS Compact Trie	511
22.9	Trie Structure	512
23.1	Example of floyd's cycle finding	532
24.1	Subsequence Problems Listed on LeetCode	556
24.2	Interval questions	575
24.3	One-dimensional Sweep Line	576
24.4	Min-heap for Sweep Line	577
25.1	Example of insertion in circular list	590
25.2	Histogram	593
25.3	Track the peaks and valleys	599
25.4	profit graph	601
25.5	Task Scheduler, Left is the first step, the right is the one we end up with.	603
26.1	LPS length at each position for palindrome.	610

27.1 Example of Binary search tree of depth 3 and 8 nodes.	620
27.2 The lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.	620
27.3 Illustration of Segment Tree.	628
27.4 Trie VS Compact Trie	631
27.5 Trie Structure	631
29.1 State Transfer Tree Structure for LIS, each path represents a possible solution. Each arrow represents an move: find an element in the following elements that's larger than the current node.	665
29.2 Word Break with DFS. For the tree, each arrow means check the word = parent-child and then recursively check the result of child.	667
29.3 Caption	672
29.4 One Time Graph Traversal. Different color means different levels of traversal.	679
29.5 Caption	682
29.6 Caption	684
29.7 Tree Structure for One dimensional coordinate	688
29.8 Longest Common Subsequence	690
29.9 Caption	706
30.1 Copy process	727
30.2 Caption	728
30.3 Caption	729
30.4 Caption	729

List of Tables

3.1	10 Main Categories of Problems on LeetCode, total 877 . . .	38
3.2	Problems categorized by data structure on LeetCode, total 877	40
3.3	10 Main Categories of Problems on LeetCode, total 877 . . .	41
9.1	Common Methods of String	116
9.2	Common Boolean Methods of String	116
9.3	Common Methods of List	119
9.4	Common Methods for Sequence Data Type in Python	125
9.5	Common out of place operators for Sequence Data Type in Python	125
9.6	Common Methods of Deque	138
9.7	Datatypes in Queue Module, maxsize is an integer that sets the upperbound limit on the number of items that can be places in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.	139
9.8	Methods for Queue's three classes, here we focus on single-thread background.	139
9.9	Methods of heapq	163
10.1	Analog of Asymptotic Relation	182
11.1	Performance of Search Algorithms on Trees or Acyclic Graph	208
14.1	Methods of bisect	280
15.1	Comparison operators in Python	305
15.2	Operator and its special method	307

16.1 Tabulation VS Memoization	352
27.1 Time complexity of operations for BST in big O notation	626
29.1 Different Type of Single Sequence Dynamic Programming	652
29.2 Different Type of Coordinate Dynamic Programming	652
29.3 Process of using prefix sum for the maximum subarray	658
29.4 Different Type of Coordinate Dynamic Programming	677
30.1 Arithmetic operators in Python	720
30.2 Comparison operators in Python	721
30.3 Logical operators in Python	721
30.4 Identity operators in Python	722
30.5 Membership operators in Python	722
30.6 Special Methods for Object Creation, Destruction, and Representation	726
30.7 Special Methods for Object Creation, Destruction, and Representation	726
30.8 Container Data types in collections module.	733

0

Preface

Preface

Graduating with a Computer science or engineering degree? Converting from physics, or math, or any unrelated field to computer science? Dreaming of getting a job as a software engineer in game-playing companies such as Google, Facebook, Amazon, Microsoft, Oracle, LinkedIn, and so on? Unfortunately, there are the most challenging “coding interview” guarding the door to these top-notch tech companies. The interview process can be intimidating, with the interviewer scrutinizing every punch of your typing or scribbling on the whiteboard. Meanwhile, you are required to express whatever is on your mind to walk your interviewer through the design and analysis process and end the interview with clean and supposedly functional code.

What kind of weapons or martial arts do we need to toughen ourselves up so that we can knock down the “watchdog” and kick it in? By weapons and martial arts, I mean books and resources. Naturally, you pull out your first or second year college textbook *Introduction to Algorithms* from bookshelf, dust it off, and are determined to read this 1000-plus-pages massive book to refresh your brain with data structures, divide and conquer, dynamic programming, greedy algorithm and so on. If you are bit more knowledgeable, you would be able to find another widely used book—*Cracking the Coding Interviews* and online coding websites—LeetCode and LintCode—to prepare. How much time do you think you need to put in? A month? Two months? Or three months? You would think after this, you are done with the interview, but for software engineers, it is not uncommon to switch companies frequently. Then you need to start the whole process again until you gain a free pass to “coding interviews” via becoming an experienced senior engineer or manager.

I was in the exact same shoes. My first war started in the fall of 2015, continued for two months and ended without a single victory. I gave up the whole interview thing until two years ago when my life (I mean finances)

situation demanded me to get an internship. This time, I got to know LeetCode and started to be more problem and practice driven from the beginning. ‘Cause God knows how much I did not want to redo this process, I naturally started to dig, summarize or create, and document problem-patterns, from sources such as both English and Chinese blogs, class slides, competitive programming guideline and so on.

I found I was not content with just passing interviews. I wanted to seek the *source* of the wisdom of algorithmic problem solving—the principles. I wanted to reorganize my continuously growing knowledge in algorithms in a way that is as clear and concise as possible. I wanted to attach math that closely relates to the topic of algorithmic problem solving, which would ease my nerves when reading related books. But meanwhile I tried to avoid getting too deep and theoretical which may potentially deviate me from the topic and adds more stress. All in all, we are not majoring in math, which is not ought to be easy; we use it as a practical tool, a powerful one! When it comes to data structures, I wanted to connect the *abstract* structures to real Python objects and modules, so that when I’m using data structures in Python, I know the underlying data structures and their responding behaviors and efficiency. I felt more at ease seeing each particular algorithm explained with the source principle of algorithm design—*why* it is so, instead of treating each as a standalone case and telling me “what” it is.

Three or four months in midst of the journey of searching for answers to the above “wantes”, the idea of writing a book on this topic appeared in my mind. I did not do any market research, and did not know anything about writing a book. I just embarked on the boat, drifted along, and as I was farther and deeper in the ocean of writing the book, I realized how much work it can be. If you are reading this sometime in the future, then I landed. The long process is more of an *agile* development in software engineering; knowledge, findings, and guidelines are added piece by piece, constantly going through revision. Yet, when I started to do research, I found that there are plenty of books out there focusing on either teaching algorithmic knowledge (*Introduction to Algorithms*, *Algorithmic Problem Solving*, etc) or introducing interview processes and solving interview problems(*Cracking the Coding Interview*, *Coding Interview Questions*, etc), but barely any that combines the two. This book naturally makes up this role in the categorization; learning the algorithmic problem solving by analyzing and practicing interview problems creates a reciprocal relationship—creating passion and confidence to make $1+1=4$.

What’s my expectation? First, your feeling of enjoyment when reading and practicing along with the book is of the upmost importance to me. Second, I really wish that you would be able to sleep well right the night before the interview which proves that your investment both financially and timewise was worthwhile.

In all, this is a book that unites the algorithmic problem solving, Coding

Interviews, and Python objects and modules. I tried hard to do a good job. This book differs from books focusing on extracting the exact formulation of problems from the fuzzy and obscure world. We focus on learning the principle of algorithm design and analysis and practicing it using well-defined classical problems. This knowledge will also help you define a problem more easily in your job.

Li Yin

Li Yin
<http://liyinscience.com>
8/30/2019

Acknowledgements

1

Reading of This Book

1.1 Structures

I summarize the characteristics that potentially set this book apart from other books seen in the market; starting from introducing technically what I think of the core principles of algorithm design are—the “source” of the wisdom I was after as mentioned in the preface, to illustrating the concise organization of the content, and to highlighting other unique features of this book.

Core Principles

Algorithm problem solving follows a few core principles: Search and Combinatorics, Reduce and Conquer, Optimization via Space-Time Trade-off or be Greedy. We specifically put these principles in one single part of the book—Part. IV.

1. In Chapter. IV (Search and Combinatorics), I teach how to formulate problems as searching problems via combinatorics in the field of math to enumerate its state space—solution space or all possibilities. Then we further optimize and improve the efficiency through “backtracking” techniques.
2. In Chapter. ??(Reduce and Conquer) we can either reduce problem A to problem B (solving problem B means we solved problem A) or Self-Reduction to reduce problem to a series of subproblems (Such as these algorithm design methodologies fall into this area: divide and conquer, some search algorithms, dynamic programming and greedy algorithms). **Mathematical induction** and **recurrence relations**

play as an important role in problem solving, complexity analysis and even correctness proof.

3. When optimization is needed, we have potentially two methods: when we see the subproblems/states overlap, space-time trade-off can be applied such as in dynamic programming; Or we can make greedy choice based on current situation.

Concise and Clear Organization

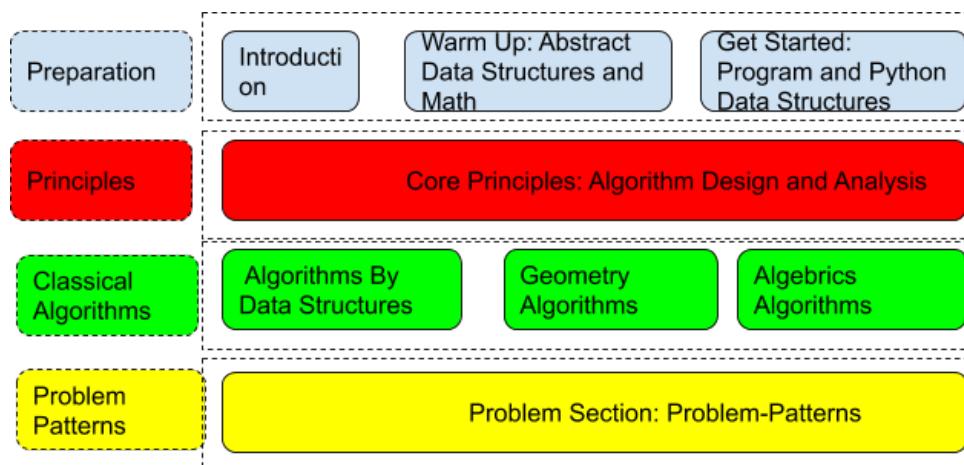


Figure 1.1: Four umbrellas: each row indicates corresponding parts as outlined in this book.

In this book, we organize in the ordering of Part, Chapter, Section, Subsection, Subsubsection and Paragraph. The parts will be categorized under four umbrellas and each serves an essential purpose:

1. Preparation: Introduce the global picture of algorithmic problem solving and coding interviews, learn abstract data structures and highly related and useful math such as recurrence relation, and hands-on Python practice by relating the abstract data structures to Python data structures.
2. Principles: As we introduced in the core principle part, we organize the design and principle here so that readers can use them as guidance while not seeking for peculiar algorithm for solving a problem.
3. Classical algorithms: We enhance our algorithm database via learning how to apply the core principles to a variety of classical problems. A database that we can quickly relate to when seeing problems.

4. Coding interview problem patterns: We close our book with the analyzing and categorizing problems by patterns. We address classical and best solutions for each problem pattern.

Other Features and Uniqueness

1. The exercise and answer setting: at the problem-pattern section, the first chapter will be named problem pool which list all problems with description. At each exercise section across chapters, only problem id is referred. Instead the answers to problems are organized by different patterns so that users can review problem solving skills quickly when preparing for an interview. This is also practical to problem solving skills.
2. Real coding interview problems referred from LeetCode, users can easily practice online and join discussions with other users.
3. Real Python Code included in the textbook and offered via Google Colab instead of using Pseudo-code.
4. The content is grain-scaled, great for users to skim when necessary to prepare for interviews.
5. Included practical algorithms that are extremely useful for solving coding interview problems and yet are almost never be included in other books, such as monotone stack, two-pointer techniques, and bit manipulation with Python.
6. Included highly related math methods to ease the learning of the topic, including recurrence relations, math formulas, math induction method.
7. Explanation of concepts are problem solving oriented, this makes it easier for users to grasp the concepts. We introduce the concepts along with examples, we strengthen and formalize the concepts in the summary section.

Q & A

What do we not cover? In the spectrum of coding interviews and the spectrum of the algorithms, we do not include:

- Although this book is a comprehensive combination of Algorithmic Problem Solving and Coding Interview Coaching, I decided not to provide preparation guideline to the topic of **System Design** to avoid deviation from our main topic. An additional reason is, personally, I have no experience yet about this topic and meanwhile it is not a topic that I am currently interested in either, so a better option is to look for that in another book.

- On the algorithm side, we briefly explain what is **approximate algorithms**, **heuristic search**, and linear programming, which is mainly seen in Artificial Intelligence, such as machine learning algorithms and neural networks. We do mention it because I think it is important to know that the field of artificial intelligence are just simply a subfield of algorithms, it is powerful because of its high dimensional modeling and large amount of training data.

How much we include about Python 3? We use Python 3 as our programming language to demonstrate the algorithms for its high readability and popularity in both industry and academics. We mainly focus on Python built-in Data types, frequently used modules, and a single class, and leave out knowledge such as object-oriented programming that deals with class heritages and composition, exception handling, and so on. Our approach is to provide brief introduction to any prior Python 3 knowledge when it is first used in the book, and put slightly more details in the Appendix for further reading and reference. We follow PEP 8 Python programming style. If you want to the object-oriented programming in Python, Python 3 Object-oriented programming is a good book to use.

Problem Setting Compared with other books that talk about the problem solving (e.g. *Problem Solving with Algorithms and Data Structures*, we do not talk about problems in complex setting. We want the audience to have a simple setting so that they can focus more on analyzing the algorithm or data structures' behaviors. This way, we keep out code clean and it also serves the purpose of coding interview in which interviewees are required to write simpler and less code compared with a real engineering problems because of the time limit.

Therefore, the purpose of this book is three-fold: to answer your questions about interview process, prepare you fully for the “coding interview”, and the most importantly master algorithm design and analysis principles and sense the beauty of them and in the future to use them in your work.

1.2 Reading Suggestions

We divide the learning of this book in four stages, each stage builds up on each other. Evaluate which stage you are, and we kindly suggest you to read in these orders:

- **First Stage** I recommend readers first start with Part Two, fundamental algorithm design and analysis, part Three, bit manipulation and data structures to know the basics in both algorithm design and data structures. In this stage, for graph data structures, we learn BFS and DFS with their corresponding properties to help us understand

more graph and tree based algorithms. Also, DFS is a good example of recursive programming.

- **Second Stage** In the second stage, we move further to Part Four, Complete Search and Part Five, Advanced Algorithm Design. The purpose of this stage is to move further to learn more advanced algorithm design methodologies: universal search, dynamic programming, and greedy algorithms. At the end, we will understand under what condition, we can improve our algorithms with efficiency from searching-based algorithms to dynamic programming, and similarly from dynamic programming to greedy algorithms.
- **Third Stage** After we know and practiced the universal algorithm design and know their difference and handle their basic problems. We can move to the third stage, where we push ourselves further in algorithms, we learn more advanced and special topics which can be very helpful in our career. The content is in Part Six, Advanced and Special Topics.
 1. For example, we learn more advanced graph algorithms. They can be either BFS or DFS based.
 2. Dynamic programming special, where we explore different types of dynamic programming problems to gain even better understanding to this topic.
 3. String pattern Matching Special:
- **Fourth Stage** In this stage, I recommend audience to review the content by topics:
 1. Graph: Chapter Graphs, Chapter Non-linear Recursive Backtracking, Chapter Advanced Graph Algorithms, Chapter Graph Questions.
 2. Tree: Chapter Trees, Chapter Tree Questions
 3. String matching: Chapter String Pattern Matching Special, Chapter String Questions
 4. Other topics: Chapter Complete Search, Chapter Array Questions, Chapter Linked List, Stack, Queue and Heap.

Wanna Finish the Book ASAP? Or just review it for interviews?

I organize the book in all of forty chapters, it is a lot but they are carefully put under different parts to highlight each individual's purpose in the book. We can skim difficult chapters marked by asterisk(*) that will unlikely appear in a short-time interview. The grained categorization helps us to skim

on the chapter levels, if you are confident enough with some chapters or you think they are too trivial, just skim, given that the book is designed to be self-contained of multiple fields(programming languages, algorithmic problem solving and the coding interview problem patterns).

The content within the book is almost always partitioned into paragraphs with titles. This conveniently allows us to skip parts that are just for enhancement purpose, such as “stores” or . This helps us skim within each chapter.

Part I

Introduction

2

The Global Picture of Algorithmic Problem Solving

“We problem modeling with data structures and problem solving with algorithms, Data structures often influence the details of an algorithm. Because of this the two often go hand in hand.”

– Niklaus Wirth, *Algorithms + Data Structures = Programs*, 1976

In this chapter, we build up a global picture of algorithmic problem solving to guide the reader through the whole “ride”.

2.1 Introduction

In the past, a person who is capable of solving complex math/physics computation problem faster than the ordinary stands out and is highly sought after. For example, during world war two, Alan Turing hired engineer who was fast solving the Sudoku problems. These kind of stories die with the rise of powerful machines, with which the magic sticks are handed over to ones—programmers who are able to harness the continually growing computation power of the hardwares to solve those once only a handful or none of people that can solve, with algorithms.

There are many kinds of programmers. Some of them code the real-world, obvious and easy rules to implement applications, some others challenge more computational problems with knowledge in math, calculus, geometry, physics, and so. We give a universal definition of algorithmic problem solving—information processing. Three essential parts include: Data structures, algorithms, and programming languages. Knowing some basic data structures, some types of programming languages and some basic algorithms are enough for the first type of programmers. They might focus

16.2. THE GLOBAL PICTURE OF ALGORITHMIC PROBLEM SOLVING

more on the front-end, such as mobile design, webpage design. The second type of programmers, however, need to be equipped with more advanced data structures and algorithm design and analysis techniques. Sadly, it is all just a start, the real powerful lie in the combination of these algorithm design methodologies and the other subjects. Math among all is the most important, for both design and analysis, as we will see in this book. Still a candidate with strong algorithmic skills is off a good start, at least with some basic math knowledge, we can almost always manage to solve problems with brutal force searching, and some others with dynamic programming.

Let us continue to define the algorithmic problem solving as information processing, just **what** it is, and not **how** at this moment.

2.1.1 What?

Introduction to Data Structure Information is the data we care about, which needs to be structured. And we can think of data structure as our low-level file manager, what it needs to do is to support four basic operations—'find' a file belongs to Bob, 'Add' Emily's file, 'Delete' Shown's file, 'Modify' Bod's file. Why structured? If you are the file manager, would you just throw all the hundreds of files over the floor or just throwing over in the drawer? Nope, you line them up in the drawer, or you even put a name on top of each file and order them by their first name. The way data is structured in program is similar to real-world system, simply lining up, or organize like a tree structure if there is some belonging and hierarchical ordering which appears in institutions and companies.

Introduction to Algorithms Algorithms further process data with a series of basic operations—searching, modifying, inserting, deleting, and so—that come with input data's structures or even auxiliary data's structures if necessary. How to design and analyze this series of operations are the field of algorithmic problem solving.

Same problem can be solved with different level of complexities in time and storing space. Deep down, algorithm designers

With this information processing step, we get our task done—computing our high school math, sorting the student ids in order, searching a word in a document, you name it.

Programming Language A programming language especially higher level of language such as Python would come with data structures that might already have the basic operations: search, modify, insert, delete. For example, the `list` module in Python, it is used to store an array of items, it comes with `append()`, `insert()`, `remove`, `pop` that you can operate your data, thus, a `list` can be viewed as a data structure. If we know what data structure we save our input instance, what algorithms to use to operate the data,

we can code these rules with a certain programming language and let the computer take over and if it won't demands billions of operations, it will get the result way more faster than humans are capable of, this is why we need computers anyway.

2.1.2 How?

Knowing what it is, now, you would ask how. How can we know how to organize our data, how to design and analysis our algorithm? how to program it? We need to study existing and well-designed data structures, algorithm design principle and algorithm analysis techniques, understand and analyze our problems, and study classical algorithms that our predecessors invented for solving a classical problem, only then when we are seeing a problem, old or new, we are prepared, we compare it with problems we know how to solve: if it is exact the same same, congratulations, we would solve our problem; if it is similar to a certain category of problems, at least we start from a direction and not from scratch; if it is totally new, at least we have our algorithm design principle and analysis techniques, we design one after understanding the problem and relate it to all our skills. Of course, there are problems that no body has been able to solve it yet. We will study it in the book so that you would identify when the problem you are solving is too hard.

The Tree of Algorithmic Problem Solving Back to the question, how? We study and build up our knowledge and skill base. A well-organized and explained knowledge base will surely ease our nerves and make things easier. The field of algorithms and computer science is highly flexible. Assuming the knowledge of computer science is a tree, and assume that each leaf is a specific algorithm to solve a specific type of problem. What would be the root of the tree? The main trunk, branches? It is impossible for us to check or even count the number of leaves. But, we can understand the tree by knowing its structures. This book is fascinated with this belief and shows a lot of effort into organizing the algorithm design and analysis methodologies, data structures, and problem patterns. It starts with the rooting algorithm design and analysis principle, and we study classical leaves by relating it and explained with our principle rather than treating each one individually.

The algorithm design and analysis principles comprise the trunk of the algorithm tree. A branch would be applying a type of algorithm design principle on a certain type of data structure, for example, algorithms related to tree structures, to graph structures, to string, to list, to set, to stack, to priority queue and so on.

18.2. THE GLOBAL PICTURE OF ALGORITHMIC PROBLEM SOLVING

2.1.3 Organization of the Contents

Based on our understanding of what is algorithmic problem solving and how to solve it, we organize the content of the book as:

- Part ?? includes the abstract commonly used data structures in computer science, the math tools for design, correctness prove, and some geometry knowledge that we need to solve geometry problems that are still often seen in the interviews.
- Part ?? strengthens the programming skills by implementing data structures and some basic coding.
- Part ?? is our main trunk of the algorithmic programming solving.
- Part ?? to Part ?? takes us to different branches and showcases classical algorithms within that branch. One or many algorithm design principles can be applied to solve these problems.
- Part. ?? is the problem patterns. Actually, if we have a good grasp of the sections before, this section is more of a review and exercises section. The finding of the patterns are to ease our coding interview preparation.

As a part of the introduction part, As I always believe, setting up the big picture should be the very first part of any technical book; it helps to know how each part plays its role global-wise with more details comes from the preface. The organization of this chapter follows the global picture and each element of algorithmic problem solving is further briefed on in each section:

- Problem Modeling (Section. 2.3), includes Data structures, hands-on examples.
- Problem Solving (Section. 2.4), includes Algorithm Design and Analysis Methodologies (Section. 2.4.2) and Programming Language(Section. 2.5).

2.2 Introduction

Algorithms are Not New Algorithms should not be considered purely abstract and obscure. It originates from real-life problem solving including time before there even exist computers (machines). The recurrence were studied as early as 1202 by L. Fibonacci, for whom the Fibonacci number is named. Algorithms, as a set of rules/actions to solve a problem, they leverage any form of knowledge – math, physics. Math stands out among all, as it is our tool to understand problems, present relations, solve problems, and analyze complexity. In this book, we use math in the most practical way and only

at places where it really matters. The difference is, with computer program written in a certain computer language to execute the algorithm is way more efficient generally than doing it in person.

Algorithms are Everywhere in our daily life. Assume you are given a group of people, your task is to find if there is a person in the group that is born on a certain day. The most intuitive way to do is to check each of them and see if his/her birthday matches with the target, this needs you to go a full-round of this group of people. If you observed that this group of people is grouped by the months, then you can nail down the times of checking by checking the subgroup that matches the month of your target day. The first way is the easiest and most straightforward way to solve a problem, which is called brute force. The second one is involves more observation and might takes less time to get the answer. However, they both have one thing in common, need us to nail down the possibilities; in the first way, we nail it down one by one, and in the second, we nail it down by almost $11/12$ of the original possibility. We can say solving the problem is to find its solution in solution space, and different way of finding the solution is called different algorithm.

2.3 Problem Modeling

The very first thing is to find or be given a problem exist in the world and solving it can bring practical value and hopefully make some good effect on the mother natural or humanity. In problem modeling, we analyze the characteristics of problem and relate it on certain data structures.

In the stage of problem modeling, we define the problem and model our problems with data structures. In this section, we first answer the question, “what is a problem in computer science?” Then, we introduce the “skeleton” –Data Structures to prepare for our next step –problem solving. Then, we give hands-on Examples about how to model a problem with data structures.

If you are a zoologist, these are how you define a species: describe the fresh and appearance, put together its skeletons, search similar well-studied species from dataset, match observed behaviors, and induce the unknown ones from similar species. There are two key steps to problem modeling:

1. Understand the problems (Section. 2.3.1): We give the definition of problems, followed by the problem categories which categorizing problems without the context of data structures. This is like describe the fresh of a species
2. Apply data structures to the problems (Section. 2.3.2): We then describe our problem in terms of data structures; connecting the fresh and the skeletons. We also analyze the the problem by exploring its

20.2. THE GLOBAL PICTURE OF ALGORITHMIC PROBLEM SOLVING

solution space and simulating the process; finding the series of actions between the input and output instance.

2.3.1 Understand Problems

Problem Formulation

A problem can be a task or something to be done according to the definition of “problem” in English dictionary, such as finding a ball numbered 11 from a pool of numbered balls, sorting the list of students by their IDs. The first thing we need to understand should be *problem formulation* and the closest knowledge we need to define a problem comes from the field of math. The intuitive definition of a problem is that it is a set of related tasks, usually infinite.

The formal definition of problems: A problem is characterized by:

1. **A set of input instances:** The *instance* represents some real examples of this type of problems. And input instances are data, which needed to be saved and accessed from the machine. This mostly requires us to define a data structure, however, different data structures can be used to define.
2. **A task to be preformed on the input instances:** The problem definition should usually comes with examples to better explain how the task decides the output of the exemplary input instances.

For example, we formulate the problem of drawing a call from the pool as: Given a list of unsorted integers, find if the number 11 is in the list, return true or false.

Example :

```
Given the list : [1, 34, 8, 15, 0, 7]
Return False because 11 does not appear in the list .
```

Problem Categories

Now, to better understand what computer science deals with, we categorize problems commonly solved in the field.

Continuous or Discrete? Based on whether the variables are continuous or discrete, we have two categories of problems:

1. **Continuous problems:** relates to continuous solution spaces.
2. **Discrete problems:** relates to discrete solution spaces.

The field of algorithmic problem solving is highly correlated to *Discrete Mathematics*, which covers topics such as arithmetic and geometric sequence, recurrence relations, inductions, graph theory, generating functions, number theory, combinatorics, and so. Through this book. some important parts are detailed (recurrence relation, induction, combinatorics, graph theory) which serves as powerful tools to do good job in computer science.

What do They Ask? We may be asked to answer four types of questions:

1. **YES/No Decision Problems:** answering whether a number is prime, odd or even are examples of such decision problems.
2. **Search problems:** Find one/all *feasible solutions* that meets problem requirement, which requires the identification of a solution from within a potentially infinite set of possible solutions. For example, finding the n^{th} prime number. Almost all problems are or can be converted to a search problem in some way. Further, search problems can be divided into:
 - **Counting Problems:** Count all feasible solutions to a search problem, such as answering, ‘how many of the 100 integers are prime?’.
 - **Optimization Problems:** Find the *best solution* among all feasible solutions to a search problem. In addition to the search problem, optimization problems answers the decision, ‘is the solution the best among all feasible ones?’.

Combinatorics When discrete problems are asked with counting or optimization questions, in computer science we further have combinatorial problems, which is also widely called *combinatorics*.

Combinatorics originates from discrete mathematics and become part of computer science. As the name suggested, combinatorics is about combining things; it answers questions: "How many ways can these items be combined?", and "Whether a certain combination is possible, or what combination is the ‘best’ in some sense?"

Through this book, permutations, combinations, subsets, strings, points in the linear order, and trees, graphs, polygons in the non-linear ordering will be examined (suggest contents in the book). We will have some briefy study on this topic in Chapter ??.

Tractable or Intractable? The complexity of a problem is normally described in relation with the size of the input instance. If a problem is algorithmic and computable, being able to produce a solution may depend on the size of the input or the limitations of the hardware used to implement

22.2. THE GLOBAL PICTURE OF ALGORITHMIC PROBLEM SOLVING

it. Based on if a problem can be possibly solved by existing machines we have:

1. **Tractable problems:** If a problem has reasonable solution, that it can be solved in no more than polynomial time complexity, it is said to be tractable.
2. **Intractable problems:** Some problems can only be solved with algorithms whose execution time grows too quickly in relation to their input size, say exponential, then these problems are considered to be intractable. For example, the classical Traveling Salesperson Problem.

Problems can also be categorized as:

1. **P Problems:**
2. **NP Problems:**

There are more types, such as *undecidable problems* and *the halting problems*, feel free to look them up if interested.

2.3.2 Understand Solution Space

A data structure is a specialized format of organizing, storing, processing, and retrieving data. As Dr. Wirth states in the chapter quote, we problem modeling with data structures and the data structures often influence the details of an algorithm: the input/output instances, and the intermediate results in the process of an algorithms all associates to data structures.

In this section, we do not intend to get into details of data structures, but rather pointing out directions. Quickly skim the first section of Part. ?? and get a sense of the categories of data structures. When a problem is modeled with data structures, the problems can further be classified based on its data structures. At this stage, we should try to model our input on a data structure, and analyze the following five components to even better understand our problem.

Five Components There are generally five components of a problem that we can define and depends on to correlate the problem to data structures, and to algorithms—searching, divide and conquer, dynamic programming, and greedy algorithms. We introduce the five components with a dummy example:

Given a list of items $A=[1, 2, 3, 4, 5, 6]$, find the position of item with value 4.

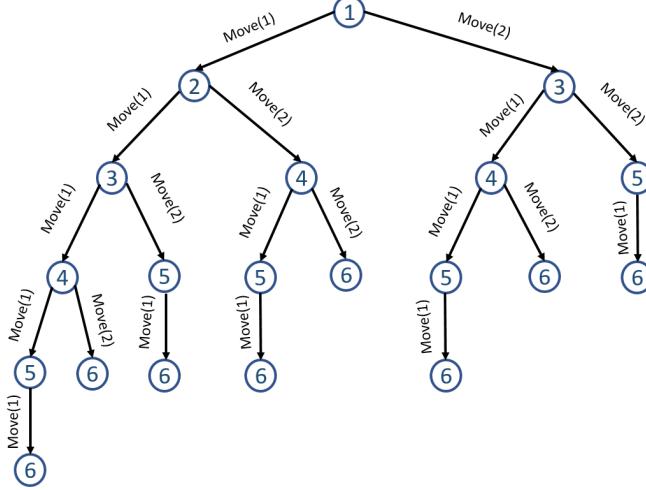


Figure 2.1: The State Space Graph. This may appears as a tree, but we can redraw it as a graph.

1. **Initial State:** state that where our algorithm starts. In our example, we can scan the whole list starting from leftmost position 0, we denote it as $S(0)$. Note that a state does not equal to a point on the input instance, it can be a range –such as from position 0 to 5, or from 0 to 2, or any state you define.
2. **Actions or MOVES:** describe possible actions relating to a state. Now, given position 1 with 2 as value, we can move only one step forward and get to position 2 or we can move 2, 3, 4, 5 steps and so. Thus, we should find all possible actions or moves that we can take to progress to next state. We can denote it as $\text{ACTIONS}(1)=\text{MOVE}(1)$, $\text{MOVE}(2)$, $\text{MOVE}(3)$, $\text{MOVE}(4)$, $\text{MOVE}(5)$.
3. **State transfer Model:** decides the state results from doing an action a at state s . We denote it as $T(a, s)$. For example, if we are at position 1 and move one step, $\text{MOVE}(1)$, then we can reach to state 2, which can be denote as $2 = T(\text{MOVE}(1), 1)$.
4. **State Space:** is the set of all states reachable from the initial state by any sequence of actions, in our case, it can be 0, 1, 2, 3, 4, 5. We can infer state space of the problem from the initial state, actions, and transfer model. The state space forms a directed network or *graph* in which the nodes are states and the links between nodes are actions. Graph, with all its flexibility, is a universal and natural way to represent relations. For example, if we limit the maximum moves we can make at each state to two, the state space will be formed as follows in Fig. 2.6. In practice, draw the graph as a tree structure is another option; in the

24.2. THE GLOBAL PICTURE OF ALGORITHMIC PROBLEM SOLVING

tree, we observe repeat states due to the expansion of nodes in graph with multiple ingoing links. A *path* in the state space is a sequence of states connected by a sequence of actions.

5. **Goal Test:** the determines whether a given state is a goal state. Such as in this example, the goal state is 4. The goal is not limited to such enumerated sets of states, it can also be specified by an abstract property. For example, in the constraint state problems(CSP) such as the n-queen, the goal is to reach to a state that not a single pair of queens will attack each other.

In this example, the space graph is an analysis tool; we use it to represent the transition relationship between different states not the exact data structure that we use to operate and define algorithms on.

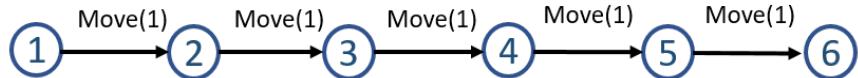


Figure 2.2: State Transfer process on a linear structure

Apply Data Structures With the state space graph, our problem is abstracted to finding a node with value 4 and graph algorithms—more specifically, graph search—can be applied to solve the problem. It does not take an expert to tell us, "This graph just complicated the situation, because our intuition can lead us to a much simpler and straightforward solution: scan the items from the leftmost to the rightmost one by one". True! As is depicted in Fig. 2.2, the problem can be modeled using a linear structure, possibly a list or linked list, and we only need to consider one action out of all options, MOVE(1), then our searching covers the whole state space, which makes the algorithm we designed *complete*¹. On the other side, in the state space graph, if we insist on moving two steps each time, we would not be able to cover the whole state space, and might end up not finding our target, which indicates this algorithm is *incomplete*.

Instead of using linear data structure, we can restructure the states as a tree if we refine the state as a range of items. The initial state is the possible subarray the target can be found, denote as $S(0, 5)$. Start from initial state, each time, we divide the space into two halves: $S(s, m)$ and $S(m, e)$, where s, e is the start and end index respectively, and $m = (s + e) // 2$, meaning the integer part of $s + e$ divided by 2. We do this to all nodes repeatedly, and we will have another state transfer graph shown in Fig. 2.3. From this graph we can see, the last node will be where we can not divide further, that is

¹Check complexity analysis

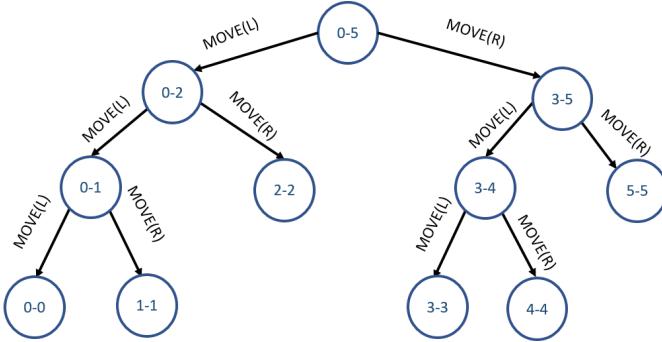


Figure 2.3: State Transfer Process on the tree

when $s = e$. From state $0 - 5$ to $3 - 5$ needs an action—move to the right. Similarly, from $0 - 5$ to $0 - 3$ needs the action of moving to the left. We use $\text{MOVE}(L)$, $\text{MOVE}(R)$ to denote the whole set of possible actions to take.

In this example, we showed how same simple problem can be modeled using two different data structures—linked list and tree.

2.4 Problem Solving

In this section, we will first demonstrate how algorithm can be applied on these two data structures with its corresponding state transfer process. Following this, we introduce the four fundamental algorithm design and analysis methodologies—the “soul/brain”. We end this section by briefing on categorizing algorithms.

2.4.1 Apply Design Principle

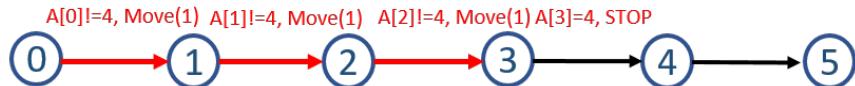


Figure 2.4: Linear Search on explicit linear data structure

Given the state transfer graph in Fig. 2.2, we simply iterate each state and compare each item with our target to see if it equals; if true, we find our target and return, if not, we continue to the end. This simple search method is depicted in Fig. 2.4. What if we know that the data is already organized in ascending order? With the tree data structure, when given a specific target, we only need to choose one action from the actions set; either move to left or right to search with a condition: if target is larger or smaller than the item in the middle of the state. When 4 is target, we have the search process depicted in Fig. 2.5.

26 2. THE GLOBAL PICTURE OF ALGORITHMIC PROBLEM SOLVING

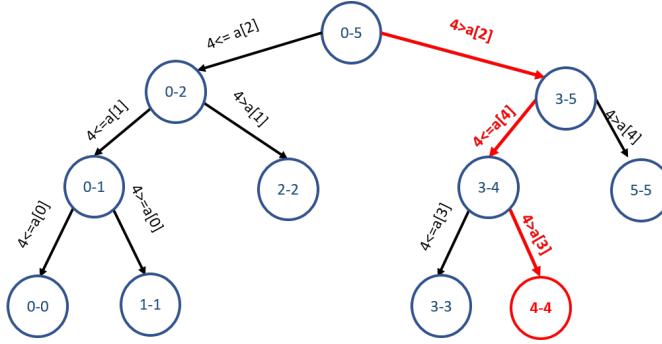


Figure 2.5: Binary Search on an implicit Tree Structure

All these state space, data structure, algorithm, and analysis might appear overwhelming to you for now. But as you learn, some of these steps are not necessary, but knowing these elements are good for you to analyze and learn new algorithms, think of it more gathering terminologies into your language base.

2.4.2 Algorithm Design and Analysis Principles

Algorithm Design

More of the time, the most naive and inefficient solution – *brute-force solution* would strike us right away, which is simply searching a feasible solution to the problem in its solution space using the massive computation power of the hardware. Although the naive solution is not preferred by your boss nor it will be incorporated into the real product, it offers the baseline for your complexity comparison and to showcase how good your well-designed algorithm is.

In the dummy example, we actually used two different searching algorithms—linear search and binary search. The process of looking for a sequence of actions that reaches the goal is called search. Therefore, *searching* is the fundamental strategy and the very first step to problem-solving. How could it not be? Algorithms are about to find answers to problems, if and assuming we can define our potential state/solution space, then a naive/exhaustive searching would do the magic and solve the problem. However, back to reality, we are limited by computation resource and speed, we comprise by:

1. being smarter than we can be to decrease the cost, increase the speed, and yet still give out the exact solution we are looking for. This comes down to *optimization*, which we have *divide and conquer*(Chapter ??), *dynamic programming*(Chapter ??), and *greedy algorithms*(Chapter ??). What are the commonalities between them? They all in some way need

us to get *recurrence relation*((Chapter ??), which is essentially *mathematical induction*(Chapter ??), which I generalized from another book, *Introduction to Algorithms: A Creative Approach*, by Udi Manber. Explain it in another way, these principles are using recurrence relation to find the relation of a problem with its smaller instance. Why is it smarter? First, smaller problems are just easier to solve than larger problems. Second, the cost of assembling the answer to smaller problems to answers to the larger problems is possibly smaller.

2. by approximating the answer. Instead of trying to get the exact answer, we find one that is good enough. Here goes to all heuristic search, machine learning, artificial intelligence. Guess, currently my limited knowledge is not enough for me to give more context that this.

Equally, we can say all algorithms can be described and categorized as searching algorithms. Yet, there are three algorithm design paradigms—*Divide and Conquer*, *Dynamic Programming*, and *greedy Algorithms*, can be applied in the searching process for faster speed or using less space. Don't worry, this is just the introduction chapter, all these concepts and algorithm design principles will be explained later.

Algorithm Analysis of Performance

How to measure problem-solving performance? Up till now, we have some basic ways to solve the problem, we need to consider the criteria that might be used to measure them. We can evaluate an algorithm's performance in four ways:

1. **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
2. **Optimality:** Does the strategy find the optimal solution, as defined?
3. **Time Complexity:** How long does it take to find a solution?
4. **Space Complexity:** How much memory is needed to perform the search?

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state space graph, $|V| + |E|$, where V is the set of vertices (nodes) of the graph and E is the set of edges (links). This is appropriate when the graph is an explicit data structure that is input to the search program. However, in reality, it is better to describe the search tree that applied to search for our solutions. For this reason, complexity can be expressed in terms of three quantities: b , the **branching factor** or a maximum number of successors of any node; d , the **depth** if the shallowest goal

28.2. THE GLOBAL PICTURE OF ALGORITHMIC PROBLEM SOLVING

node ; and m , the maximum length of any path in the state space. Time is often measured in terms of the number of nodes in the search tree, and the space are in terms of the maximum number of nodes stored in memory.

For the most part we describe time and space complexity for search on a tree; for a graph, the answer depends on how “redundant” paths or “loops” in the state space are.

2.4.3 Algorithm Categorization

There are countless algorithms invented, however, these traditional data-independent algorithms (not the current data-oriented deep learning models which are trained with data), it is important for us to be able to categorize the algorithms and understand the similarities and characteristics of each type and also be able to compare each type:

- By implementation: the most useful in our book is recursive and iterative. Understand the difference of these two, and the special usage of recursion (Chapter III) is fundamental to the further study of algorithm design. We can also have serial and parallel/distributed, deterministic and non-deterministic algorithms. In our book, all the algorithms we learn are serial and deterministic algorithms.
- By design: algorithms can be interpreted to one or several of the four fundamental problem solving paradigms, Divide and Conquer (Part ??), Dynamic Programming and Greedy (Part ??). In Section ??, we will briefly introduce and compare these four problem solving paradigms to gain a global picture of the spirit of algorithms.
- By complexity: mostly algorithms can be categorized by its time complexity. Given an input size of n , we normally have categories of $O(1)$, $O(\log n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, and $O(n!)$. More details and the comparison is given in Section ??.

The intractable problems are still get solved by computer. We can limit our input instance size. However, it is not very practical, when the size of the input size is large and we are still hoping to get solutions, maybe not the best, but are good enough in a reasonable(polynomial) time. *Approximate algorithms* comes into our hand, such as *heuristic algorithm*. In this book, we focus more on the non-approximate algorithmic methods to solve problems in *discrete* solution spaces, and only brief on the part of approximate algorithms.

2.5 Programming Languages

hird, for certain type of problems, there are algorithms specifically designed and tuned to optimize that type of question. This wil be introduced in

Part ???. Which might give us almost the best efficiency we can find.

2.6 Tips for Algorithm Design

Principle

1. Understand the problem, analyze with searching and combinatorics to get the complexity of the naive solution.
2. If it is a exponential problem, check if the dynamic programming applies. If not, we have to stick to a search algorithm. If it applies, we can decrease the complexity to polynomial.
3. If it is polynomial already or polynomial after the dynamic programming applied, check if the greedy approach or the divide and conquer can be applied to further decrease the polynomial complexity. For example, if it is $O(n^2)$, divide and conquer might decrease it to $O(n \log n)$, and the greedy approach might end up with $O(n)$.
4. If none of these design principle applies: we stick to the searching and try to optimize with better searching techniques—such as backtracking, bidirectional search, A^* , sliding window and so on.

This process can be generalized with “BUD”—bottleneck, unnecessary work, and D.

2.7 Exercise

2.7.1 Knowledge Check

Longest Increasing Subsequence

 Practice first before you check up the solution. (put the solution at next page)

Given a list of items $A = [1, 2, 3, 4, 5, 6]$, find the position of item with value 4.

1. **Initial State:** state that where our algorithm starts. In our example, we can scan the whole list starting from leftmost position 1. $S(0)$
2. **Actions or MOVES:** A description of possible actions available at a state. If we are at position 1, we can have different possible actions, we can move only one step forward and get to position 2. Or we can move 2, 3, 4, 5 steps. We can denote it as $\text{ACTIONS}(1)=\text{MOVE}(1)$, $\text{MOVE}(2)$, $\text{MOVE}(3)$, $\text{MOVE}(4)$, $\text{MOVE}(5)$.

30.2. THE GLOBAL PICTURE OF ALGORITHMIC PROBLEM SOLVING

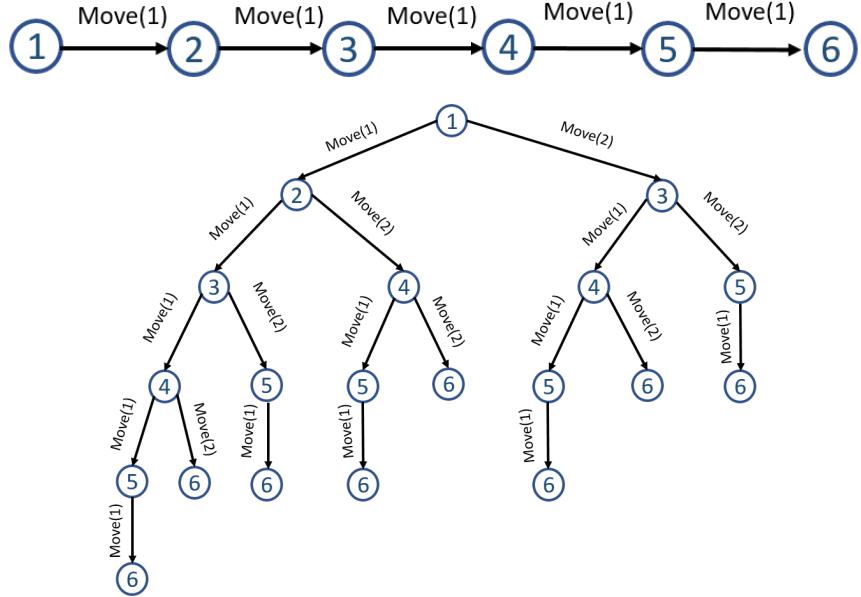


Figure 2.6: The State Spaces Graph

3. **State transfer or transition model:** It returns the state results from doing an action a at state s . We denote it as $T(a, s)$. For example, if we are at position 1 and take action that move one step, MOVE(1), then we can reach to state 2, denote as $2 = T(MOVE(1), 1)$. We also use the term **successor** to refer to any state reachable from a given state by a single action.
4. **State Space:** Together, the initial state, actions, and transition model implicitly define the state space of the problem—the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed network or **graph** in which the nodes are states and the links between nodes are actions. For example, if we limit the maximum moves we can make at each state to be one and two, the state space will be formed as follows in Fig. 2.6. A **path** in the state space is a sequence of states connected by a sequence of actions.
5. **Goal Test:** the goal test determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. Such as in this example, the goal state is 4. Sometimes the goal is specified by an abstract property rather than explicitly enumerated sets of states. For example, in the constraint state problems(CSP) such as the n-queen, the goal is to reach to a state that not a single pair of queens

will attack each other.

In practice, analyzing and solving a problem is not answering a yes or no question. There are always multiple angles to model a problem, the way to model and formalize a problem decides the corresponding algorithm that can be used to solve this problem. And it might also decide the efficiency and difficulty to solve the problem. For example, using the Longest Increasing Subsequence: **Ways to model the problem.** There are different ways to

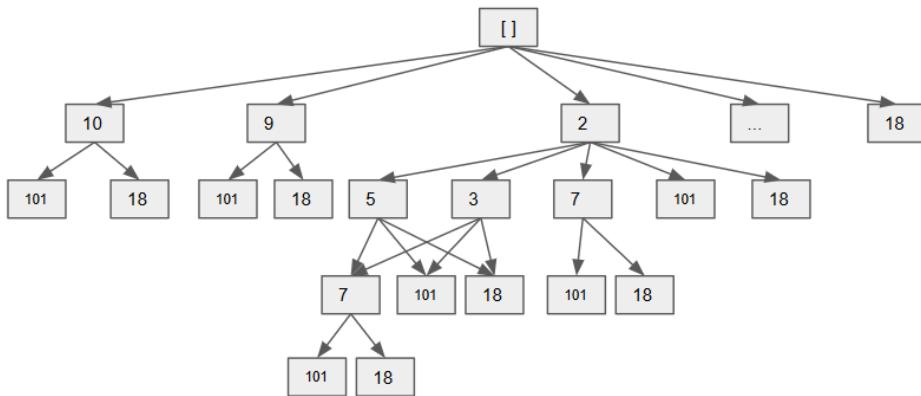


Figure 2.7: State Transfer Tree Structure for LIS, each path represents a possible solution. Each arrow represents a move: find an element in the following elements that's larger than the current node.

model this LIS problem, including:

1. Model the problem as a directed graph, where each node is the elements of the array, and an edge μ to v means node $v > \mu$. The problem now becomes finding the longest path from any node to any node in this directed graph.
2. Model the problem as a tree. The tree starts from empty root node, at each level i , the tree has $n-i$ possible children: $\text{nums}[i+1], \text{nums}[i+2], \dots, \text{nums}[n-1]$. There will only be an edge if the child's value is larger than its parent. Or we can model the tree as a multi-choice tree: for combination problem, each element can either be chosen or not chosen. We would end up with two branch, and the nodes would become a path of the LIS, therefore, the longest LIS exist at the leaf nodes which has the longest length.
3. Model it with divide and conquer and optimal substructure.

32 2. THE GLOBAL PICTURE OF ALGORITHMIC PROBLEM SOLVING

3

Coding Interviews and Resources

In my humble opinion, I think it is a waste of our precious time to read either books or long blogs that purely focusing on the interview process and preparation. Personally, I would rather read a book that amuses me or work on my personal project that has some meanings or just enjoy it with your friends and families.

This chapter consists of parts:

1. Tech Interviews (Section. 3.1)
2. Tips and Resources on Coding Interviews (Section. ??).

3.1 Tech Interviews

In this section, a brief introduction to the coding interviews and hiring process for a general software engineering position is first provided. , coding interviews related with data structures and algorithms are necessary. Your masterness of such knowledge varies as the requirement of more specific work.

3.1.1 Coding Interviews and Hiring Process

Coding interviews, a.k.a whiteboard coding interviews, is one part of the whole interview process, where interviewees would be asked to solve one or a few well-defined problems and write down the code in 45-60 minutes of time window while the interviewer is watching. This process can be done either remotely via a shared file between interviewer and interviewee or face-to-face in a conference room on the whiteboard with the interviewer being present.

Typically, the interview pipeline of software developer jobs consists of three stages—*Exploratory chat with recruiter*, *Screening interviews*, and *On-site interviews*:

- Exploratory chat with recruiters: Either you applied for the position and passed the initial screening or luckily get found by recruiters, they would contact you to schedule a short chat, normally through phone. During the phone call, the recruiter would introduce the company, the position, and ask for your field of interest; just to check the degree of interest on either side and decide if the process should be continued
- Screening interviews: The screening interviews are usually two back-to-back coding interviews, each lasts 45-60 minutes. This process consists of the introduction on each side—interviewer and interviewee—which is cut as short as possible to save enough time for the coding interviews.
- On-site interviews: If you have passed the first two rounds of interviews, you would be invited to the on-site interviews which is the most fun, exciting, but might also be the most daunting and tiring part of the whole process, since they can last anywhere from four hours to the entire day. The company would offer both transportation and accommodation to get you there. The on-site interview consists of 4 to 6 rounds one-on-one, each with an engineer in the team and lasts between 45-60 minutes; due to the long process, typically a lunch interview is included. There are some extra cases, which may or may not be included: group presentation, recruiter conversation, or conversation with the hiring manager or team manager. Presentation might happens to research scientist or higher-level positions. The onsite interview appears to be more more diverse compared with screening interview; introduction, coding interviews, brain teaser type of questions, behavior questions, and questions related to the field of the position, such as machine learning or web development. During the lunch interview, it was just hanging out with the one who is arranged to be with you, chatting while eating and showing you around the company in some cases.

In some cases, you get to have to do on-line assignment which happens more to some start-ups and second tier tech companies, which requires you spending at least two hours solving problems without any promise that would lead to real interviews. Personally, I have done that twice with companies such as ; and I never heard back from them then. I fully resent such assignment; it is unfair because it wasted my time but not the company's, I learned nothing and the process is bored to hell! Ever since then I decide to stop the interview whenever such chore is demanded!

Both the first and the second process serves as an initial screening process, the purpose is obviously; a trade-off because of the cost, because the remaining interview process can be quite costly in terms of finance—accommodation and transportation if you get the on-site interviews—and in terms of time—the time cost on each side but mainly the cost from spending 5-8 hours on the interviewees from multiple engineers of the hiring company.

Sometimes the process differs slightly between internship and full-time position; interns typically do not need on-site interviews. For new graduates, getting an internship first and through the internship to get a full-time offer can ease the whole job hunting process a bit. For more experienced engineers, they might get invited to on-site without the screening.

3.1.2 Why Coding Interviews?

The History

Coding interviews originated in the form of in-person interview and writing code on paper back in 1970s, populated as whiteboard interview in 1990s with the rise of the Internet, and froze in time and continued to live till today, 2020s.

Back in the 70s, computer time was expensive; the electricity, data processing and disc space costs were outrageous. Here shows the cost per megahertz from 1970 to 2007, courtesy off Dr. Mark J. Perry:

Computer Prices, Computer Speed and Cost/MHz				
	1970	1984	1997	2007
Cost	\$4,600,000	\$4,000	\$1,000	\$550
Speed (MHz)	12.5	8.3	166	1600
Cost per MHz	\$368,000	\$482	\$6	\$0.34

Figure 3.1: Computer Prices, Computer Speed and Cost/MHz

Writing code on paper was a common, natural and effective way for programmers to code into computer later in this era. As Bill Gates describes the experience in a commencement speech at his alma mater – Lakeside School: “*You had to type up your program off-line and create this paper tape—and then you would dial up the computer and get on, and get the paper in there, and while you were programming, everybody would crowd around, shouting: “Hey, you made a typing mistake” “Hey, you messed this up” “Hey, you’re taking too much time.”*

Writing code is conducted on whiteboard rather than on paper in 1990s, when software engineering was growing exponentially with the rise of the

Internet. It was a natural transition because the whiteboard is easy to setup, erase the mistake and the entire team can see the code which makes it a perfect way to conduct discussion.

Now, at the 21st centry, when the computation is virtually free, the act of writing out the code on whiteboard or paper continues and part of our interview process.

Discussion

Is it a good way to testify and select talented candidates? There are different opinions, in all, either favors it or oppose it. Stating the reason on each side is boring and not bear much value. Let us see what people say about it in their words.

ME : How do you think about coding interviews?

SUSIE CHEN : Ummmmmm well there was like one full month I only did Leetcode before interviewing with Facebook. LOL, was a bad experience but worth it hahahah. Susie was an intern from Facebook, with bachelor degree from University of.

.....

ME : How the coding interview plays its role for new graduates and experienced engineers ?

ERIC LIN :

- Common: Both require previous proj demo/desc. Your work matters more than your score. Ppl care more about the actual experience than the paper work.
- Diffs: Grads are more asked for a passion or altitude of learning and problem-solving. For experienced engineers, the coding interview doesn't matter at all.

Eric is an Cloud Engineer at Contino, Four years of experience, Master's degree in Information Technology at Monash University, Australia.

3.2 Tips and Resources

Because of the focus of the book—learning computer science while having fun and the opinion I hold to coding interviews decide I will check this section short and offer more general information and tips.

3.2.1 Tips

Tips for Preparation

1. First and of the most important tip: Do not be afraid of applying! Apply any company that you want to join and try your best to make it in the interviews. No need to check out statistics about their hiring ratio and be terrified of trying. You have nothing to lose and it would be a good chance to get first-hand interview experience with them, which might help you next time. So, be bold, and just do it!
2. Schedule your interviews with companies in a descending order of your favoritism. This way you get as much practice as possible before you go on with your dream company.
3. Before doing real interviews, do mocking interviews; either ask your friends for help or find online mocking websites.
4. If you are not fluent in English, practice even more using English! This is the situation for a lot of international STEM students, including me.

I wished I would know the last three tips when I first started to prepare interview with Google back to 2016 (At least I followed the first tip, went for my dream company without hesitation, huh), one year in my PhD. It was a very first try, I prepared for a month(I mean at least 8 hours a day); reading and finishing all problems from *Cracking the Coding Interview*. I failed the screening interview that is conducted through the phone with a Google share document. I was super duper nervous; taking long to just understand the question itself given my poor speaking English that time and the noise from the phone made the situation worse (talking with people on the phone fears me more than the ghost did from *The Shining*, by Stephen King). At that time, I also did not have a clue about LeetCode.

5 Tips for the Interview Here, we summarize five tips when we are doing a real interview or trying to mock one beforehand in the preparation.

1. Identify Problem Types Quickly: When given a problem, we read through the description to first understand the task clearly, and run small examples with the input to output, and see how it works intuitively in our mind. After this process, we should be able to identify the type of the problems. There are 10 main categories and their distribution on the LeetCode which also shows the frequency of each type in real coding interviews.
2. Do Complexity Analysis: We brainstorm as many solutions as possible, and with the given maximum input size n to get the upper bound of time complexity and the space complexity to see if we can get AC while not LTE.

For example, For example, the maximum size of input n is 100K, or 10^5 ($1K = 1, 000$), and your algorithm is of order $O(n^2)$. Your common

Table 3.1: 10 Main Categories of Problems on LeetCode, total 877

Types	Count	Ratio1	Ratio 2	
Ad Hoc	Array String			
Complete search	Iterative Search Recursive Search	84 43	27.8% 22.2%	15.5% 13.6%
Divide and Conquer	15	8%	4.4%	
Dynamic Programming	114	6.9%	3.9%	
Greedy	38			
Math and Computational Geometry	103	3.88%	2.2%	
Bit Manipulation	31	2.9%	1.6%	
Total	490	N/A	55.8%	

sense told you that $(100K)^2$ is an extremely big number, it is 10^{10} . So, you will try to devise a faster (and correct) algorithm to solve the problem, say of order $O(n \log_2 n)$. Now $10^5 \log_2 10^5$ is just 1.7×10^6 . Since computer nowadays are quite fast and can process up to order 1M, or 10^6 (1M = 1, 000, 000) operations in seconds, your common sense told you that this one likely able to pass the time limit.

3. Master the Art of Testing Code: We need to design good, comprehensive, edges cases of test cases so that we can make sure our devised algorithm can solve the problem completely while not partially.
4. Master the Chosen Programming Language:

3.2.2 Resources

Online Judge System

Leetcode LeetCode is a website where you can practice on real interviewing questions used by tech companies such as Facebook, Amazon, Google, and so on.

Here are a few tips to navigate the usage of LeetCode:

- **Use category tag to focusing practice:** With the category or topic tags, it is better strategy to practice and solve problems one type after another, shown in Fig. 3.2.
- **Use test case to debug:** Before we submit our code on the LeetCode, we should use the test case function shown in Fig. 3.4 to debug and testify our code at first. This is also the right mindset and process at the real interview.
- **Use Discussion to get more solutions:**

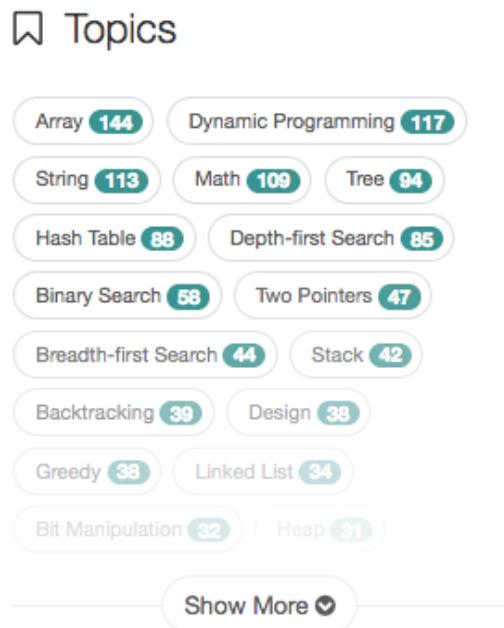


Figure 3.2: Topic tags on LeetCode



Figure 3.3: Use Test Case to debug

Algorithm Visualizer If you are inspired more by visualization, then check out this website, <https://algorithm-visualizer.org/>. It offers us a tool to visualize the running process of algorithms.

Mocking Interviews Online

Interviewing.io Use the website interviewing.io, you can have real mocking interviews given by software engineers working in top tech company. This can greatly help you overcome the fear, tension. Also, if you do well in the practice interviews, you can get real interviewing opportunities from their partnership companies.

Interviewing is a skill that you can get better at. The steps mentioned above can be rehearsed over and over again until you have fully internalized them and following those steps become second nature to you. A good way to practice is to find a friend to partner with and the both of you can take turns to interview each other.

A screenshot of a LeetCode course schedule page titled "630. Course Schedule III". The page displays a list of solutions for a specific problem. Each solution entry includes a user icon, the title, a brief description, the creation date, and the number of votes and views. The solutions listed are:

- A simple C solution 44ms (Created at: May 11, 2018 1:14 AM | No replies yet.) - 0 votes, 62 views
- Sort in lgn, DP using same method as Find Longest Increasing Subsequence (Created at: April 9, 2018 6:15 AM | No replies yet.) - 0 votes, 206 views
- Simple Python code using priority queue with explanation (Created at: November 1, 2017 7:36 AM | No replies yet.) - 0 votes, 205 views
- C++ priority queue solution (Created at: October 8, 2017 11:40 PM | No replies yet.) - 0 votes, 152 views

Figure 3.4: Use Test Case to debug

Table 3.2: Problems categorized by data structure on LeetCode, total 877

Data Structure	Count	Percentage/Total Problems	Percentage/Total Data Structure
Array	136	27.8%	15.5%
String	109	22.2%	13.6%
Linked List	34	6.9%	3.9%
Hash Table	87		
Stack	39	8%	4.4%
Queue	8	1.6%	0.9%
Heap	31	6.3%	3.5%
Graph	19	3.88%	2.2%
Tree	91	18.6%	10.4%
Binary Search Tree	13		
Trie	14	2.9%	1.6%
Segment Tree	9	1.8%	1%
Total	490	N/A	55.8%

A great resource for practicing mock coding interviews would be interviewing.io. interviewing.io provides free, anonymous practice technical interviews with Google and Facebook engineers, which can lead to real jobs and internships. By virtue of being anonymous during the interview, the inclusive interview process is de-biased and low risk. At the end of the interview, both interviewer and interviewees can provide feedback to each other for the purpose of improvement. Doing well in your mock interviews will unlock the jobs page and allow candidates to book interviews (also anonymously) with top companies like Uber, Lyft, Quora, Asana and more. For those who are totally new to technical interviews, you can even view a demo interview on the site (requires sign in). Read more about them [here](#).

Aline Lerner, the CEO and co-founder of interviewing.io and her team are passionate about revolutionizing the technical interview process and helping candidates to improve their skills at interviewing. She has also published a number of technical interview-related articles on the interview-

Table 3.3: 10 Main Categories of Problems on LeetCode, total 877

Algorithms	Count	Percentage/Total Problems	Percentage/Total Data Structure
Depth-first Search	84	27.8%	15.5%
Breadth-first Search	43	22.2%	13.6%
Binary Search	58	18.6%	10.4%
Divide and Conquer	15	8%	4.4%
Dynamic Programming	114	6.9%	3.9%
Backtracking	39	6.3%	3.5%
Greedy	38		
Math	103	3.88%	2.2%
Bit Manipulation	31	2.9%	1.6%
Total	490	N/A	55.8%

ing.io blog. interviewing.io is still in beta now but I recommend signing up as early as possible to increase the likelihood of getting an invite.

Pramp Another platform that allows you to practice coding interviews is Pramp. Where interviewing.io matches potential job seekers with seasoned technical interviewers, Pramp takes a different approach. Pramp pairs you up with another peer who is also a job seeker and both of you take turns to assume the role of interviewer and interviewee. Pramp also prepares questions for you, along with suggested solutions and prompts to guide the interviewee.

Communities

If you understand Chinese, there is a good community ¹ that we share information with either interviews, career advice and job packages comparison.

¹<http://www.1point3acres.com/bbs/>

Part II

Warm Up: Abstract Data Structures and Tools

We warm up our “algorithmic problem solving”-themed journey with knowing the abstract data structures—representing data, fundamental problem solving strategies—searching and combinatorics, and math tools—recurrence relations and useful math functions, which we decide to dedicate a standalone chapter to it due to its important role both in algorithm design and analysis, as we shall see in the following chapters.

4

Abstract Data Structures

(put a figure here)

4.1 Introduction

Leaving alone statements that “data structures are building blocks of algorithms”, they are just mimicking how things and events are organized in real-world in the digital sphere. Imagine that a data structure is an old-schooled file manager that has some basic operations: searching, modifying, inserting, deleting, and potentially sorting. In this chapter, we are simply learning how a file manager use to ‘lay out’ his or her files (structures) and each ‘lay out’s corresponding operations to support his or her work.

We say the data structures introduced in this chapter are *abstract* or idiomatic, because they are conventionally defined structures. Understanding these abstract data structures are like the terminologies in computer science. We further provide each abstract data structure’s corresponding Python data structure in Part. III.

There are generally three broad ways to organize data: Linear, tree-like, and graph-like, which we introduce in the following three sections.

Items We use the notion of **items** throughout this book as a generic name for unspecified data type.

Records

4.2 Linear Data Structures

4.2.1 Array

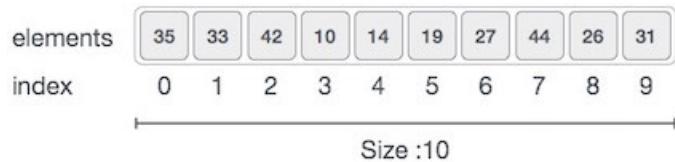


Figure 4.1: Array Representation

Static Array An array or static array is container that holds a **fixed size** of sequence of items stored at **contiguous memory locations** and each item is identified by *array index* or *key*. The Array representation is shown in Fig. 4.1. Since using contiguous memory locations, once we know the physical position of the first element, an offset related to data types can be used to access any item in the array with $O(1)$, which can be characterized as **random access**. Because of these items are physically stored contiguous one after the other, it makes array the most efficient data structure to store and access the items. Specifically, array is designed and used for fast random access of data.

Dynamic Array In the static array, once we declared the size of the array, we are not allowed to do any operation that would change its size; saying we are banned from either inserting or deleting any item at any position of the array. In order to be able to change its size, we can go for *dynamic array*. that is to say Static array and dynamic array differs in the matter of fixing size or not. A simple dynamic array can be constructed by allocating a static array, typically larger than the number of elements immediately required. The elements of the dynamic array are stored contiguously at the start of the underlying array, and the remaining positions towards the end of the underlying array are reserved, or unused. Elements can be added at the end of a dynamic array in constant time by using the reserved space, until this space is completely consumed. When all space is consumed, and an additional element is to be added, then the underlying fixed-sized array needs to be increased in size. Typically resizing is expensive because it involves allocating a new underlying array and copying each element from the original array. Elements can be removed from the end of a dynamic array in constant time, as no resizing is required. The number of elements used by the dynamic array contents is its logical size or size, while the size of the underlying array is called the dynamic array's capacity or physical size, which is the maximum possible size without relocating data. Moreover, if

the memory size of the array is beyond the memory size of your computer, it could be impossible to fit the entire array in, and then we would retrieve to other data structures that would not require the physical contiguity, such as *linked list*, *trees*, *heap*, and *graph* that we would introduce next.

Operations To summarize, array supports the following operations:

- Random access: it takes $O(1)$ time to access one item in the array given the index;
- Insertion and Deletion (for dynamic array only): it consumes Average $O(n)$ time to insert or delete an item from the middle of the array due to the fact that we need to shift all other items;
- Search and Iteration: $O(n)$ time for array to iterate all the elements in the array. Similarly to search an item by value through iteration takes $O(n)$ time too.

No matter it's static or dynamic array, they are static data structures; the underlying implementation of dynamic array is static array. When frequent need of insertion and deletion, we need dynamic data structures, The concept of static array and dynamic array exist in programming languages such as C—for example, we declare `int a[10]` and `int* a = new int[10]`, but not in Python, which is fully dynamically typed(need more clarification).

4.2.2 Linked List

Dynamic data structures, on the other hand, is designed to support flexible size and efficient insertion and deletion. Linked List is one of the simplest dynamic data structures; it achieves the flexibility by abandoning the idea of storing items at contiguous location. Each item is represented separately—meaning it is possible to have item of different data types, and all items are linked together through *pointers*. A pointer is simply a variable that holds the address of an item as a value. Normally we define a record data structure, namely `node`, to include two variables: one is the value of the item and the other is a pointer that addressing the next `node`.

Why is it a highly dynamic data structure? Imagine each node as a 'signpost' which says two things: the name of the stop and address of the next stop. Suppose you start from the first stop, you can head to the next stop since the first signpost tells you the address. You would only know the total number of stops by arriving at the end signpost, wherein no sign of the address. To add a stop, you can just put it at the end, at the head or anywhere in the middle by modifying any possible signpost before or after the one you add.

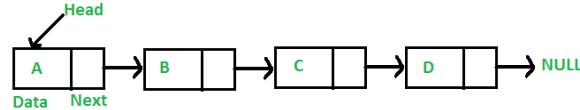


Figure 4.2: Singly Linked List

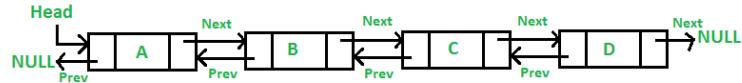


Figure 4.3: Doubly Linked List

Singly and Doubly Linked List When the node has only one pointer, it is called *singly linked list* which means we can only scan nodes in one direction; when there is two pointers, one pointer to its predecessor and another to its successor, it is called *doubly linked list* which supports traversal in both forward and backward directions.

Operations and Disadvantages

- No Random access: in linked list, we need to start from some pointer and to find one item, we need to scan all items sequentially in order to find it and access it;
- Insertion and Deletion: only $O(1)$ to insert or delete an item if we are given the node after where to insert or the node the delete.
- Search and Iteration: $O(n)$ time for linked list to iterate all items. Similarly to search an item by value through iteration takes $O(n)$ time too.
- Extra memory space for a pointer is required with each element of the list.

Recursive A linked list data structure is actually a *recursive data structure*; any node can be treated as a head node thus making it a sub-linked list.

4.2.3 Stack and Queue

Stacks and queues are **dynamic arrays** with restrictions on deleting operation. Items adding and deleting in a stack follows the “Last in, First out(LIFO)” rule, and in a queue, the rule is “First in, First out(FIFO)”,

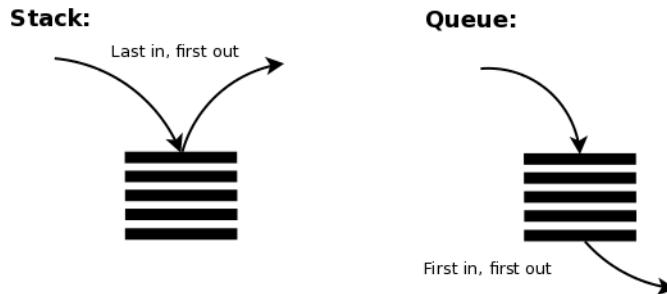


Figure 4.4: Stack VS Queue

this process is shown in Fig. 4.4. We can simply think of stack as a stack of plates, we always put back and fetch a plate from the top of the pile. Queue is just like a real-life queue in any line, to be first served with your delicious ice cream, you need to be there in the head of the line.

Implementation-wise, stacks and queues are a simply dynamic array that we add item by appending at the end of array, and they only differs with the delete operation: for stack, we delete item from the end; for a queue, we delete item from the front instead. Of course, we can also implement with any other linear data structure, such as linked list. Conventionally, the add and deletion operation is called “push” and “pop” in a stack, and “enqueue” and “dequeue” in a queue.

Operations Stacks and Queues support limited access and limited insertion and deletion and the search and iteration relies on its underlying data structure.

Stacks and queues are widely used in computer science. First, they are used to implement the three fundamental searching strategies—Depth-first, Breath-first, and Priority-first Search. Also, stack is a recursive data structure as it can be defined as:

- a stack is either empty or
- it consists of a top and the rest which is a stack;

4.2.4 Hash Table

A hash table is a data structure that (a) stores items formed as {key: value} pairs, (b) and uses a *hash function* $index = h(key)$ to compute an index into an array of buckets or slots, from which the mapping value will be stored and accessed; for users, ideally, the result is given a key we are expected to find its value in constant time—only by computing the hash function. An example is shown in Fig. 4.5. Hashing will not allow two pairs that has the same key.

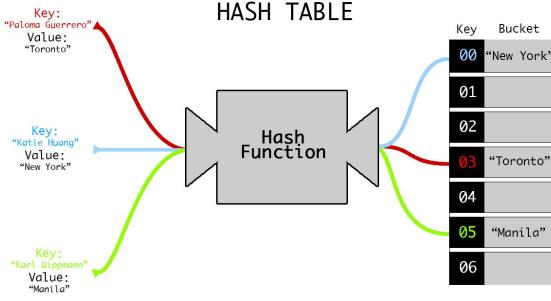


Figure 4.5: Example of Hashing Table, replace key as index

First, the key needs to be of real number; when it is not, a conversion from any type it is to a real number is necessary. Now, we assume the keys passing to our hash function are all real numbers. We define a *universe* set of keys $U = \{0, 1, 2, \dots, |U - 1|\}$. To frame hashing as a math problem: given a set of keys drawn from U that has n {key: value} pairs, a hash function needs to be designed to map each pair to a key in a set in range $\{0, \dots, m - 1\}$ so that it fits into a table with size m (denoted by $T[0 \dots m - 1]$), usually $n > m$. We denote this mapping relation as $h : U \rightarrow \{0, \dots, m - 1\}$. The simplest hashing function is $h = \text{key}$, called *direct hashing*, which is only possible when the keys are drawn from $\{0, \dots, m - 1\}$ and it is usually not the case in reality.

Continue from the hashing problem, when two keys are mapped into the same slot, which will surely happen given $n > m$, this is called *collision*. In reality, a well-designed hashing mechanism should include: (1) a hash function which minimizes the number of collisions and (2) a efficient collision resolution if it occurs.

Hashing Functions

The essence of designing hash functions is uniformity and randomness. We further use $h(k, m)$ to represent our hash function, which points out that it takes two variables as input, the key as k , and m is the size of the table where values are saved. One essential rule for hashing is if two keys are equal, then a hash function should produce the same key value ($h(s, m) = h(t, m)$, if $s = t$). And, we try our best to minimize the collision to make it unlikely for two distinct keys to have the same value. Therefore our expectation for average collision times for the same slot will be $\alpha = \frac{n}{m}$, which is called **loading factor** and is a critical statistics for design hashing and analyze its performance. Besides, a good hash function satisfied the condition of simple uniform hashing: each key is equally likely to be mapped to any of the m slots. But usually it is not possible to check this condition because one rarely knows the probability distribution according to which the keys

are drawn. There are generally four methods:

1. **The Direct addressing method**, $h(k, m) = k$, and $m = n$. Direct addressing can be impractical when n is beyond the memory size of a computer. Also, it is just a waste of spaces when $m \ll n$.
2. **The division method**, $h(k, m) = k \% m$, where $\%$ is the module operation in Python, it is the remainder of k divided by m . A large prime number not too close to an exact power of 2 is often a good choice of m . The usage of prime number is to minimize collisions when the data exhibits some particular patterns. For example, in the following cases, when $m = 4$, and $m = 7$, keys = [10, 20, 30, 40, 50]

	m = 4	m = 7
10	$10 = 4 * 2 + 2$	$10 = 7 * 1 + 3$
20	$20 = 4 * 5 + 0$	$20 = 7 * 2 + 6$
30	$30 = 4 * 7 + 2$	$30 = 7 * 4 + 2$
40	$40 = 4 * 10 + 0$	$40 = 7 * 5 + 5$
50	$50 = 4 * 12 + 2$	$50 = 7 * 7 + 1$

Because the keys share a common factor $c = 2$ with the bucket size $m = 4$, this will decrease the range of the remainder into m/c of its original range. As shown in the example, the remainder is just $\{0, 2\}$ which is only half of the space of m . The real loading factor increase to $c\alpha$. Using a prime number is a easy way to avoid this since a prime number has no factors other than 1 and itself.

If the size of table cannot easily to be adjusted to a prime number, we can use $h(k, m) = (k \% p \% m$, where p is our prime number which should be chosen from the range $m < p < |U|$.

3. **The multiplication method**, $h(k, m) = \lfloor m(kA \% 1) \rfloor$. $A \in (0, 1)$ is a chosen constant and a suggestion to it is $A = (\sqrt{5} - 1)/2$. $kA \% 1$ means the fractional part of kA which equals to $kA - \lfloor kA \rfloor$. It is also shorten as $\{kA\}$. For example, the fractional part of 45.2 is .2. In this case, the choice of m is not as critical as in the division method; for convenience, it is suggested with $m = 2^p$, where p is some integer.
4. **Universal hashing method**: because any fixed hash function is vulnerable to the worst-case behavior when all n keys are hashed to the same index, an effective way is to choose the hash function randomly from a set of predefined hash functions for each execution—the same hash function must be used for all accesses to the same table. However, finding the predefined hash functions requires us to define multiple prime numbers if the division method for each function is used, which is not easy. A replacement is to define $h(k, m) = ((ak + b)\%p)\%m$, $a, b < p$, a, b are both integers.

Resolving Collision

Collision is unavoidable given that $m < n$ and the sometimes it just purely bad luck that the data you have and the chosen hashing function produce lots of collisions, thus, we need mechanisms to resolve possible collisions. We introduce three methods: Chaining, Open Addressing, and Perfect Hashing.

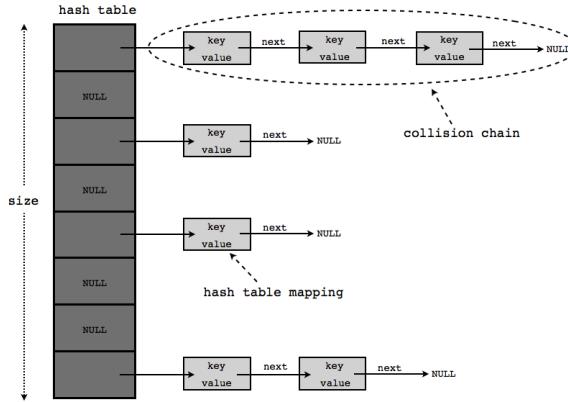


Figure 4.6: Hashtable chaining to resolve the collision, change it to the real example

Chaining An easy way to think of is by chaining the keys that have the same hashing value using a linked list (either singly or doubly). For example, when $h(k, m) = k \% 4$, and keys = [10,20,30,40,50]. For key as 10, 30, 50, they are mapped to the same slot 2. Therefore, we chain them up at index 2 using a single linked list shown in Fig. 4.6. This method shows the following characters:

- The average-case time for searching is $O(\alpha)$ under the assumption of simple uniform hashing.
- The worst case running time for insertion is $O(1)$.
- the worst-case behavior is when all keys are mapped to the same slot.

The advantage of chaining is the flexible size because we can always add more items by chaining behind, this is useful when the size of the input set is unknown or too large. However, the advantage comes with a price of taking extra space due to the use of pointers.

Open Addressing In Open addressing, all items are stored in the hash table itself; thus requiring the size of the hash table to be ($m \geq n$), making each slot either contains an item or empty and the load factor $\alpha \leq 1$. This

avoids the usages of pointers, saving spaces. So, here is the question, what would you do if there is collision?

- Linear Probing: Assume, at first, from the hash function, we save an item at $h(k_1, m) = p_1$, when another pair $\{k_2, v_2\}$ comes, we have index $h(k_2, m)$. If the index is the same as k_1 , we can simply check the position right after p_1 in a cyclic order(from p_1 to the end of hash table, continue from the start of the table and end at $p_1 - 1$): if it is empty, we save the value at $p_1 + 1$, otherwise, we try $p_1 + 2$, and so on until we find an empty spot, this is called *linear probing*. However, there are other keys such as k_3 that is mapped to index p_1 at the first time too, for k_3 , it would collide with k_1 at p_1 , with k_2 at $p_1 + 1$, and the second collision is called *secondary collision*. When the table is relative full, such secondary collision can degrade the searching in the hashing table to linear search. We can denote the linear probing as

$$h'(k, i, m) = (h(k, m) + i) \% m, \text{for } i = 0, 1, 2, \dots, m - 1. \quad (4.1)$$

i marks the number of tries. Now, try to delete item from linear probing table, we know that $T[p_1] = k_1, T[p_1 + 1] = k_2, T[p_1 + 2] = k_3$. say we delete k_1 , we repeat the hash function, find and delete it from p_1 , working well, we have $T[p_1] == \text{NULL}$. Then we need to delete or search for k_2 , we first find p_1 and find that it is empty already, then we thought k_2 is deleted, great! You see the problem here? k_2 is actually at $p_1 + 1$ but from the process we did not know. A simple resolution instead of really deleting the value, we add a flag, say `deleted`, at any position that a key is supposedly be deleted. Now, to delete k_1 , we have p_1 is marked as `deleted`. This time, when we are trying to delete k_2 , we first go to p_1 and see that the value is not the same as its value, we would know we should move to $p_1 + 1$, and check its value: it equals, nice, we put a marker here again.

- *Other Methods: However, even if the linear probing works, but it is far from perfection. We can try to decrease the secondary collisions using *quadratic probing* or *double hashing*.

In open addressing, it computes a probe sequence as of $[h(k, m, 0), h(k, m, 1), \dots, h(k, m, m-1)]$ which is a permutation of $[0, 1, 2, \dots, m-1]$. We successively probe each slot until an empty slot is found.

*Perfect Hashing

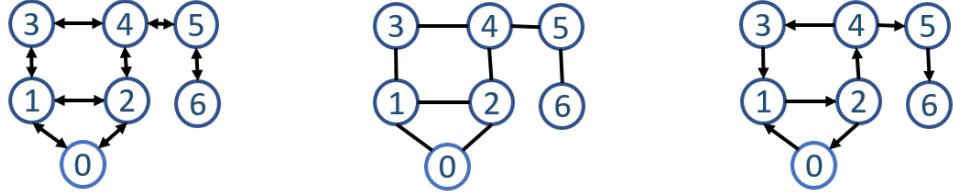


Figure 4.7: Example of graphs. Middle: undirected graph, Right: directed graph, and Left: representing undirected graph as directed, Rightmost: weighted graph.

4.3 Graphs

4.3.1 Introduction

Graph is a natural way to represent connections and reasoning between things or events. A graph is made up of *vertices* (nodes or points) which are connected by *edges* (arcs or lines). A graph structure is shown in Fig. 4.7. We use G to denote the graph, V and E to refer its collections of vertices and edges, respectively. Accordingly, $|V|$ and $|E|$ is used to denote the number of nodes and edges in the graph. An edge between vertex u and v is denoted as a pair (u, v) , depending on the type of the graph, the pair can be either ordered or unordered.

There are many fields in that heavily rely on the graph, such as the probabilistic graphical models applied in computer vision, route problems, network flow in network science, link structures of a website in social media, and so. We present graph as a data structure. However, graph is really a broad way to model problems; for example, we can model the possible solution space as a graph and apply graph search to find the possible solution to a problem. So, do not let the physical graph data structures limit our imagination.

The representation of graph is deferred to Chapter. ???. In the next section, we introduce the types of graphs.

4.3.2 Types of Graphs

Undirected Graph VS Directed Graph If one edge is directed that it points from u (the tail) to v (the arc head), but not the other way around, this means we can reach to v from u , but not the opposite. An ordered pair (u, v) can denote this edge, in this book, we denote it as $(u \rightarrow v)$. If all edges are directed, then we say it is a directed graph as shown on the right of Fig. 4.7. If all edges $e \in E$ is an unordered pair (u, v) , that it is reachable from both way for $u, v \in V$ then the graph is undirected graph as shown in the middle of Fig. 4.7.

Unweighted Graph VS Weighted Graph In *weighted* graphs, each edge of G is assigned a numerical value, or weighted. For example, the road network can be drawn as a directed and weighted graph: two edges if it is a two-way road, and one arc if its is one-way instead; and the weight of an edge might be the length, speed limit or traffic. In *unweighted* graphs, there is no cost distinction between various edges and vertices.

Embedded Graph VS Topological Graph A graph is defined without a geometric position of their own, meaning we literally can draw the same graph with vertices arranged at different positions. We call a specific drawing of a graph as an embedding, and drawn graph is called embedded graph. Occasionally, the structure of a graph is completely defined by the geometry of its embedding, such as the famous travelling salesman problem, and grids of points are another example of topology from geometry. Many problems on an $n \times m$ grid involve walking between neighboring points, so the edges are implicitly defined from the geometry.

Implicit Graph VS Explicit Graph Certain graphs are not explicitly constructed and traversed, but it can be modeled as a graph. For example, grids of points can also be looked as implicit graph, where each point is a vertex and usually a point can link to its neighbors through an implicit edge. Working with implicit graph takes more imagination and practice. Another example will be seen in the backtracking as we are going to learn in Chapter ??, the vertices of the implicit graph are the states of the search vector while edges link pair of states that can be directly generated from each other. It is totally ok that you do not get it right now, relax, come back and think about it later.

Terminologies of Graphs In order to apply graph abstractions in real problems, it is important to get familiar with the following important terminologies of graphs:

1. **Path:** A path in a graph is a sequence of adjacent vertices. For example, there is a path $(0, 1, 2, 4)$ in both the undirected and directed graph in Fig. 4.7. The length of a path in an unweighted graph is the total number of edges that it passes through—i.e., it is one less than the number of vertices in the graph. A *simple path* is a path with no repeated vertices. In the weighted graph, it may instead be the sum of the weights of all of its consisting edges. Obtaining the shortest path can be a common task and of real-value.
2. **cycles:** In directed graph a *cycle* is a path that starts and ends at the same vertex, and in undirected graph. A cycle can have length one, i.e. a *selfloop*. A *simple cycle* is a cycle that has no repeated

vertices other than the start and the end vertices being the same. In an undirected graph a (simple) cycle is a path that starts and ends at the same vertex, has no repeated vertices other than the first and last, and has length at least three. In this book we will exclusively talk about simple cycles and hence, as with paths, we will often drop simple. A graph is *acyclic* if it contains no cycles. Directed acyclic graphs are often abbreviated as **DAG**.

3. **Distance:** The *distance* $\sigma(u, v)$ from a vertex u to a vertex v in a graph G is the shortest path (minimum number of edges) from u to v . It is also referred to as the *shortest path length* from u to v .
4. **Diameter:** The *diameter* of a graph is the maximum shortest path length over all pairs of vertices: $\text{diam}(G) = \max \sigma(u, v) : u, v \text{ in } V$.
5. **Tree:** An acyclic and undirected graph is a *forest* and if it is connected it is called a *tree*. A *rooted tree* is a tree with one vertex designated as the root. A tree can be directed graph too, and the edges are typically all directed toward the root or away from the root. We will detail more in the next section.
6. **Subgraph:** A *subgraph* is another graph whose vertices V_s and edges E_s are subsets of G , and all endpoints of E_s must be included in E_s — V_s might have additional vertices. When $V = V_s, E_s \subset E$, that the subgraph includes all vertices of graph G is called **A spanning subgraph**; when $E_s = E, V_s \subset V$, that the subgraph contains all the edges whose endpoints belong the vertex subset is called **an induced subgraph**.
7. **Complete Graph:** A graph in which each pair of graph vertices is connected by an edge is a complete graph. A complete graph with $|V|$ vertices is denoted as $K_n = (|C|, 2) = n(n - 1)/2$, pointing out that it will have $(|V|, 2)$ total edges.

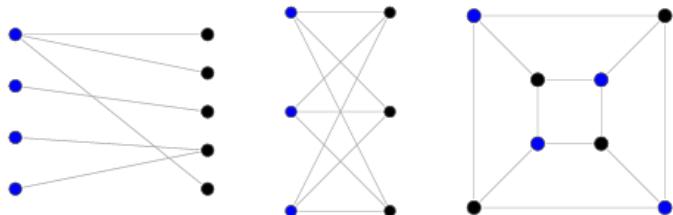


Figure 4.8: Bipartite Graph

8. **Bipartite Graph:** A bipartite graph, a.k.a bigraph, is a graph whose vertices can be divided into two disjoint sets V_1 and V_2 such that no two

vertices within the same set are connected to each other or adjacent. A bipartite graph is a graph with no odd cycles; equivalently, it is a graph that may be properly colored with two colors. See Fig. 4.8.

9. **Connected Graph:** A graph is connected if there is a path joining each pair of vertices, that it is always possible to travel in a connected graph between one vertex and any other. If a graph is not fully connected, but has subset V_s that are connected, then the connected parts of a graph are called its **components**.

4.3.3 Reference

1. <http://www.cs.cmu.edu/afs/cs/academic/class/15210-f14/www/lectures/graph-intro.pdf>

4.4 Trees

Trees in Interviews The most widely used are binary tree and binary search tree which are also the most popular tree problems you encounter in the real interviews. A large chance you will be asked to solve a binary tree or binary search tree related problem in a real coding interview especially for new graduates which has no real industrial experience and pretty much had no chance before to put the major related knowledge into practice yet.

4.4.1 Introduction

A tree is essentially a simple graph which is (1) connected, (2) acyclic, and (3) undirected. To connect n nodes without a cycle, it requires $n - 1$ edges. Adding one edge will create one cycle and removing one edge will divides a tree into two components. Trees can be represented as a graph whose representations we have learned in the last section, such a tree is called *free tree*. A **Forest** is a set of $n \geq 0$ disjoint trees.

However, free trees are not commonly seen and applied in computer science (not in coding interviews either) and there are better ways—*rooted trees*. In a rooted tree, a special node is singled out which is called the *root* and all the edges are oriented to point away from the root. The rooted node and one-way structure enable the rooted tree to indicate a hierarchy relation between nodes whereas not so in the free tree. A comparison between free tree and the rooted tree is shown in Fig. 4.9.

Rooted Trees

A rooted tree introduces a **parent-child, sibling relationship** between nodes to indicate the hierarchy relation.

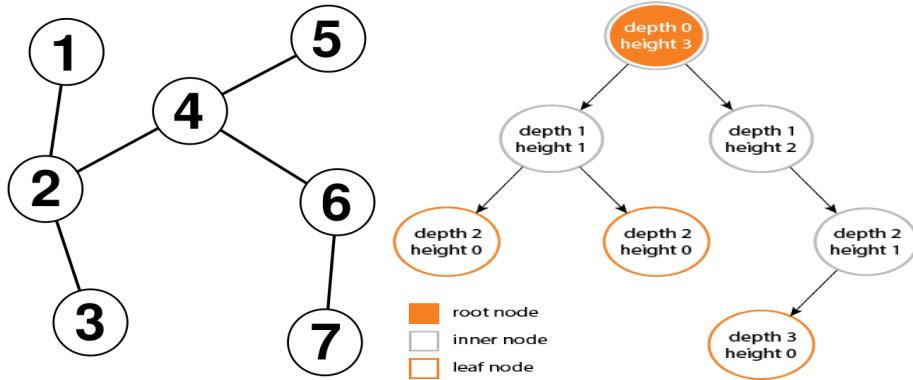


Figure 4.9: Example of Trees. Left: Free Tree, Right: Rooted Tree with height and depth denoted

Three Types of Nodes Just like a real tree, we have the root, branches, and finally the leaves. The first node of the tree is called the **root node**, which will likely be connected to its several underlying children node(s), making the root node the parent node of its children. Besides the root node, there are another two kinds of nodes: *inner nodes* and *leaf nodes*. A leaf node can be found at the last level of the tree which has no further children. An inner node is any node in the tree that has both parent node and children, which is also any node that can not be characterized as either leaf or root node. A node can be both root and leaf node at the same time, if it is the only node that composed of the tree.

Terminologies of Nodes We define the following terminologies to characterize nodes in a tree.

- **Depth:** The *depth* (or level) of a node is the number of edges from the node to the tree's root node. The depth of the root node is 0.
- **Height:** The *height* of a node is the number of edges on the *longest path* from the node to a leaf. A leaf node will have a height of 0.
- **Descendant:** The *descendant* of a node is any node that is reachable by repeated proceeding from parent to child starting from this node. They are also known as *subchild*.
- **Ancestor:** The *ancestor* of a node is any node that is reachable by repeated proceeding from child to parent starting from this node.
- **Degree:** The **degree** of a node is the number of its children. A leaf is necessarily degree zero.

Terminologies of Trees Following the characteristics of nodes, we further define some terminologies to describe a tree.

- **Height:** The *height*(or *depth*) of a tree would be the height of its root node, or equivalently, the depth of its deepest node.
- **Diameter:** The *diameter* (or *width*) of a tree is the number of nodes (or edges) on the longest path between any two leaf nodes.
- **Path:** A *path* is defined as a sequence of nodes and edges connecting a node with a descendant. We can classify them into three types:
 1. Root->Leaf Path: the starting and ending node of the path is the root and leaf node respectively;
 2. Root->Any Path: the starting and ending node of the path is the root and any node (inner, leaf node) respectively;
 3. Any->Any Path: the starting and ending node of the path is both any node (Root, inner, leaf node) respectively.

Representation of Trees Like linked list, which chains nodes together via pointers—once the first node is given, we can get hold of information of all nodes, a rooted tree can be represented with nodes consisting of pointers and values too. Because in a tree, a node would have multiple children, indicating a node can have multiple pointers. Such representation makes a rooted tree a *recursive* data structure: each node can be viewed as a root node, making this node and all the nodes that reachable from this node a subtree of its parent. This recursive structure is the main reason we separate it from graph field, and make it one of its own data structure. The advantages are summarized as:

- A tree is an easier data structure that can be recursively represented as a root node connected with its children.
- Trees can be always used to organize data and can come with efficient information retrieval. Because of the recursive tree structure, divide and conquer can be easily applied on trees (a problem can be most likely divided into subproblems related to its subtrees). For example, Segment Tree, Binary Search Tree, Binary heap, and for the pattern matching, we have the tries and suffix trees.

The recursive representation is also called *explicit* representation. The counterpart—*implicit* representation will not use pointer but with array, wherein the connections are implied by the positions of the nodes. We will see how it works in the next section.

Applications of Trees Trees have various applications due to its convenient recursive data structures which related the trees and one fundamental algorithm design methodology-Divide and Conquer. We summarize the following important applications of trees:

1. Unlike arrays and linked list, tree is hierarchical: (1) we can store information that naturally forms hierarchically, e.g., the file systems on a computer, the employee relation in at a company. (2) If we organize keys of the tree with ordering, e.g. Binary Search Tree, Segment Tree, Trie used to implement prefix lookup for strings.
2. Trees are relevant to the study of analysis of algorithms not only because they implicitly model the behavior of recursive programs but also because they are involved explicitly in many basic algorithms that are widely used.
3. Algorithms applied on graph can be analyzed with the concept of tree, such as the BFS and DFS can be represented as a tree data structure, and a spanning tree that include all of the vertices in the graph. These trees are the basis of other kind of computational problems in the field of graph.



Tree is a recursive structure, it can almost used to visualize any recursive based algorithm design or even computing the complexity in which case it is specifically called recursion tree.

4.4.2 N-ary Tres and Binary Tree

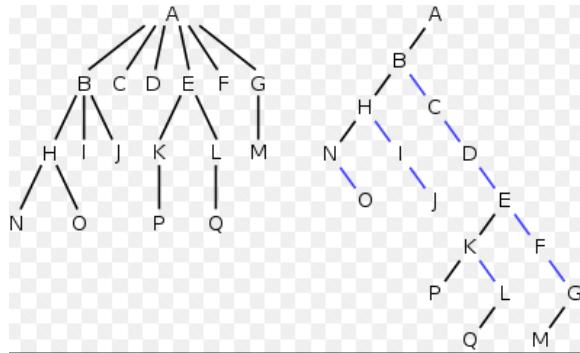


Figure 4.10: A 6-ary Tree Vs a binary tree.

For a rooted tree, if each node has no more than N children, it is called *N-ary Tree*. When $N = 2$, it is further distinguished as a *binary tree*,

where its possible two children are typically called *left child* and *right child*. Fig. 4.10 shows a comparison of a 6-ary tree and a binary tree. Binary tree is more common than N-ary tree because it is simpler and more concise, thus making it more popular for coding interviews.

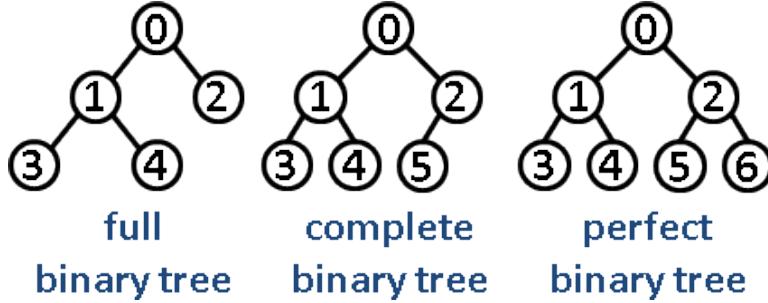


Figure 4.11: Example of different types of binary trees

Types of Binary Tree There are four common types of Binary Tree:

1. **Full Binary Tree:** A binary tree is full if every node has either 0 or 2 children. We can also say that a **full** binary tree is a binary tree in which all nodes except leaves have two children. In full binary tree, the number of leaves ($|L|$) and the number of all other non-leaf nodes ($|NL|$) has relation: $|L| = |NL| + 1$. The total number of nodes compared with the height h will be:

$$n = 2^0 + 2^1 + 2^2 + \dots + 2^h \quad (4.2)$$

$$= 2^{h+1} - 1 \quad (4.3)$$

2. **Complete Binary Tree:** A Binary Tree is **complete** if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.
3. **Perfect Binary Tree:** A Binary tree is **perfect** in which all internal nodes have two children and all leaves are at the same level. This also means a perfect binary tree is both a full and complete binary tree.
4. **Balanced Binary Tree:** A binary tree is balanced if the height of the tree is $O(\log n)$ where n is the number of nodes. For Example, *AVL tree* maintains $O(\log n)$ height by making sure that the difference between heights of left and right subtrees is at most 1.
5. Degenerate (or pathological) tree: A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

And each we show one example in Fig. 4.11.

Complete tree and a perfect tree can be represented with an array, and we assign index 0 for root node, and given a node with index i , the children will be $2 * i + 1$ and $2 * i + 2$, this is called *implicit* representation, wherein its counterpart recursive representation is called *explicit* representation.

5

Introduction to Combinatorics

In discrete optimization, some or all of the variables in a model are required to belong to a discrete set; this is in contrast to continuous optimization in which the variables are allowed to take on any value within a range of values. There are two branches of discrete optimization: integer programming and combinatorial optimization where the discrete set is a set of objects, or combinatorial structures, such as assignments, combinations, routes, schedules, or sequences. Combinatorial optimization is the process of searching for maxima (or minima) of an objective function F whose domain is a discrete but large configuration space (as opposed to an N -dimensional continuous space). Typical combinatorial optimization problems are the travelling salesman problem (“TSP”), the minimum spanning tree problem (“MST”), and the knapsack problem. We start with basic combinatorics which is able to enumerate the all solutions exhaustively. Later on, other chapters we will dive into different combinatorial/disrete optimization problems.

Combinatorics, as a branch in mathematics that mainly concerns with counting and enumerating, is a means in obtaining results, and certain properties of finite structures. Combinatorics is used frequently in computer science to obtain formulas and estimates in both the design and analysis of algorithms. It is a broad and thus seemingly hard to define topic that can solve the following types of questions:

- The counting or enumerating of specified structures, sometimes referred to as arrangements or configurations in a very general sense, associated with finite systems,
- the existence of such structures that satisfy certain given criteria, this is usually called Constraint Restricted Problems (CSPs).

- optimization, finding the “best” structure or solution among several possibilities, be it the “largest”, “smallest” or satisfying some other optimality criterion.

In this section, we introduce common combinatorics that can help us come up with the simplest which potentially be quite large state space. At least, this is the first step, and solving a small problem in this way might offer us more insights on continuing finding a better solution.

When the situation is easy, we can mostly figure out the counting with some logic and get a closed-form solution; when the situation is more complex such as in the *partition* section, we detour by using recurrence relation and math induction.

5.1 Permutation

Given a list of integer $[1, 2, 3]$, how many way can we order these three numbers? Imagine that we have three positions for these three integers. For the first position, it can choose 3 integers, leaving the second position with 2 options. Further, when it reaches to the last position, it can only choose whatever that is left, we have 1. The total count will be $3 \times 2 \times 1$.

Similarly, for n distinct numbers, we will get the number of permutation easily as $n \times (n - 1) \times \dots \times 1$. A factorial, denoted as $n!$, is used to abbreviate it. Worth to notice, the factorial sequence grows even quicker than the exponential sequence, such as 2^n .

5.1.1 n Things in m positions

Permutation of n things on n positions is denoted as $p(n, n)$. Think about what if we have $m \in [1, n - 1]$ positions instead? How to get a closed-form function for $p(n, m)$. The process is the same: we fix each position and consider the number of choice of things each one has.

$$p(n, m) = n \times (n - 1) \times \dots \times (n - (m - 1)) \quad (5.1)$$

$$= \frac{n \times (n - 1) \times \dots \times (n - m + 1) \times (n - m) \times \dots \times 1}{(n - m) \times \dots \times 1} \quad (5.2)$$

$$= \frac{n!}{(n - m)!} \quad (5.3)$$

If we want $p(n, n)$ to follow the same form, it would require us to define $0! = 1$.

 What if there are repeated things, that things are not distinct?

5.1.2 Recurrence Relation and Math Induction

The number or the full set of permutations can be generated incrementally. We demonstrate how with recurrence relation and math induction. We start from $P(0, 0) = 1$. Easily, we get $P(i, 0) = 1, i \in [1, n]$. With math induction, now assume we know $P(n - 1, m - 1)$, for the m -th position, what choice does it have? First, we need to pick this thing from the $n - (m - 1)$ things. Then, we have $m - 1$ things lined up linearly, there are m positions to insert the m -th item, resulting $P(n, m) = (n - m + 1) * mP(n - 1, m - 1)$.

Now, we can use iterative method to obtain the closed-form solution:

$$P(n, m) = (n - m + 1) * m * P(n - 1, m - 1) \quad (5.4)$$

$$= (n - m + 1) * m * (P(n - 2, m - 2)) \quad (5.5)$$

$$\dots \quad (5.6)$$

$$= m!P(n - m + 1, 1) \quad (5.7)$$

5.1.3 See Permutation in Problems

Suppose we want to sort an array of integers incrementally, say the array is $A = [4, 8, 2]$. The right order is $[2, 4, 8]$, which is trivial to obtain in this case. If we are about to form it as a search problem, we need to define a *search space*. Using our knowledge in combinatorics, we know all possible ordering of these numbers are $[4,8,2], [4,2,8], [2,4,8], [2,8,4], [8,2,4], [8,4,2]$. Generating all possible ordering and save it in an array maybe. Then this sorting problem is converted into checking which array is incrementally sorted. However, it comes with large price on space usage, since for n numbers there, the number of possible orderings are $n!$. A smarter way to do it is to check the ordering as we are generating the ordering set.

5.2 Combination

Same as before, we have to choose m things out of n but with one difference—the order does not matter, how many ways we have? This problem is called **combination**, and it is denoted as $C(n, m)$. For example, for $[1,2,3]$, $C(3, 2) = [1, 2], [2, 3], [1, 3]$. Comparatively, $P(3, 2) = [1, 2], [2, 1], [2, 3], [3, 2], [1, 3], [3, 1]$.

To get combination, we can leverage and apply permutation first. However, this results over-counting. As shown in our example, when there are two things in the combination, a permutation would double count it. If there are m things, we over count by $m!$ times. Therefore, if we divide the permutation by all permutation of m things, we get out formula for

combination:

$$C(n, m) = \frac{P(n, m)}{P(m, m)} \quad (5.8)$$

$$= \frac{n!}{(n - m)!m!} \quad (5.9)$$



Back to the last question, when there are repeats in the permutation. We can use the same idea. Assume we have n, m that n things in total but only m types and $a_i, i \in [1, m]$ to denote the number of each type, this means $a_1 + a_2 + \dots + a_m = n$. The number of ways to linearly order these objects is $\frac{n!}{a_1!a_2!\dots a_m!}$.

The combination of k things out of n , will be the same as choosing $(n-k)$ things.

$$C(n, k) = C(n, n - k) \quad (5.10)$$

5.2.1 Recurrence Relation and Math Induction

We also show how the combination can be generated incrementally. We start from $C(0, 0) = 1$, and easily we get $C(n, 0) = 1$. Assume we know $C(n - 1, k - 1)$, now we need to add the k -th item into the combination? :

- Use k -th item, then we just need to put the k -th item into any sets in $C(n - 1, k - 1)$, resulting $C(n - 1, k - 1)$.
- Not use k -th item, this means we need to pick k items from the other $n - 1$ items, resulting $C(n - 1, k)$.

Thus, we have $C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$, this is called **Pascal's Identity**.

What if things are not distinct?

5.3 Partition

We discuss three types of partitions: (1) integer partition, (2) set partition, and (3) array partition. In this section, counting becomes less obvious compared with combination and permutation, this is where we rely more on **recurrence relation** and **math induction**.

5.3.1 Integer Partition

Integer Partition Definition Integer partitions is to partition a given integer n into distinct subsets that add up to n .

```
For example, given n=5, the resulting partitioned subsets are
these 7 subsets:
{5}
{4, 1},
{3, 2}
{3, 1, 1},
{2, 2, 1},
{2, 1, 1, 1},
{1, 1, 1, 1}
```

Analysis Let us assume the resulting sequence is (a_1, a_2, \dots, a_k) , and $a_1 \geq a_2 \geq \dots \geq a_k \geq 1$, and $a_1 + a_2 + \dots + a_k = n$. The ordering is simply to help us to track the sequence. We use

The easiest way to generate integer partition is to construct them incrementally. We first start from the partition of n . For $n=5$, we get 5 first. Then we subtract one from the largest item that is larger than 1, and add it to the smallest item if it exists and that the resulting $s+1 < l$, $s < l-1$, and other option is to put it aside. For 5, there is no other item, so that it becomes 4, 1. For 4,1, following the same rule, we get 3, 2, for 3, 2, we get 3,1,1.

```
1 {5}, no other smaller item, put it aside
2 {4, 1}, satisfy s<l-1, become{3,2}
3 {3, 2}, not satisfy s<l-1, put it aside
4 {3, 1, 1}, satisfy s<l-1, add it to
5 {2, 2, 1}, not satisfy, put it aside
6 {2, 1, 1, 1}, not satisfy, put it aside
7 {1, 1, 1, 1, 1}
```



Try to generate the partition when $n=6$.

If we draw out the transfer graph, we can see a lot of overlapping of some state. Therefore, we add one more limitation on the condition, $s>1$.

5.3.2 Set Partition

Set Partition Problem Definition How many ways exist to partition a set of n distinct items a_1, a_2, \dots, a_n into $k \leq n$ nonempty subsets, $k \leq n$.

```
Here are 7 ways that we can partition the set {a1, a2, a3, a4}
into 2 nonempty subsets. They are
{a1}, {a2, a3, a4};
{a2}, {a1, a3, a4};
```

```
{a3}, {a1, a2, a4};  

{a4}, {a1, a2, a3}  

{a1, a2}, {a3, a4};  

{a1, a3}, {a2, a4};  

{a1, a4}, {a2, a3};
```

Let us denote the total ways as $s(n, k)$. As seen in the example, given 2 groups and 4 items, there are two combination of each group's size: 1+3 and 2+2. For combination 1,3, this is equivalent to choose one item from the set to put at the first subset $C(n, 1)$, and then choose 3 items for the other subset $C(3, 3)$. For combination 2, 2, we have $C(4, 2)$ for one subset and $C(2, 2)$ for the other subset. However, because the ordering of the subsets does not matter, we need to divide it by $2!$. The set partition problem thus consists of two steps:

- Partition n into k integers: This subroutine can be solved with integer partition we just learned. We have $b_1 + b_2 + \dots + b_k = n$.
- For each combination of integer partition, we compute the number of ways choosing b_i items for that set, we get $C(n, b_1) \times C(n - b_1, b_2) \times C(n - b_1 - b_2, b_3) \times \dots \times C(b_k, b_k)$. Now, we find the distinct b_i and its number of appearance in the sequence. If we have m distinct number denoted as b_i , and its count c_i , then we divide the above ways by $c_1!c_2!\dots c_m!$.

From this solution, it is hard to get a closed form for $s(n, k)$.

Find Recurrence Relation There is just one way to handle this problem, let us try the incremental method—find a recurrence relation. We first start with $s(0, 0) = 0$, and we can also easily get $s(n, 0) = 0$. Now, with the mathematical induction, we assume we solved a subproblem, say $s(n-1, k-1)$, can we induce $s(n, k)$? What do we need?

Now we have $n-1$ items in $k-1$ groups, now there is one addition group and one additional item. There are two ways:

- put the additional item into the additional group. In this way, $s(n, k)$ is simply the same as of $s(n-1, k-1)$.
- spread the $n-1$ items from the original $k-1$ groups into k groups, that is $s(n-1, k)$ and our additional item has k options now, making $k \times s(n-1, k)$ in total

Combing together the count of these two ways, we get a recurrence relation that

$$s(n, k) = s(n-1, k-1) + ks(n-1, k) \quad (5.11)$$

5.4 Array Partition

Problem Definition How many ways exist to partition an array of n items a_1, a_2, \dots, a_n into subarrays. There are different subtypes depending on the number of subarrays, say m :

1. When the number m is as flexible as $m \in [1, n - 1]$.
2. When the number m is fixed as a number in range $[2, n - 1]$.

When the number of subarray is fixed For example, it is common to partition an array into 2 or 3 subarrays. First, we find an item a_i as a partition point, getting the last subarray $a[i : n]$ and left an array to further consider $a[0 : i]$. If $m = 2$, $a[0 : i]$ results the first subarray and the partition process is done. This gives out n ways of partition. When $m = 3$, we need to further partition $a[0 : i]$ into two parts. This can be represented with recurrence relation:

$$d(n, m) = (d(i, m - 1), a[i : n]), i \in [0, n - 1] \quad (5.12)$$

Further, for $d(i, m - 1)$:

$$d(i, m - 1) = (d(j, m - 2), a[j : n]), j \in [0, i - 1] \quad (5.13)$$

This can be done recursively: we will have a recursive function with depth m .

When the number of subarray is flexible The process is the same other than m can be as large as $n - 1$. If we are about to use dynamic programming, for all these states, we need to come up with an ordering of the state (i, j) , where i is the subproblem $a[0 : i]$, and j is the number of partitions. We imagine it as a matrix with i, j as row and column respectively:

	0	1	2	...	$n-1$: partition
0	X	-	-	-	
1	X	X	-	-	
2	X	X	X		
$n-1$					
n	X	X	X	X	X

Does the ordering of the `for` loop matter? Actually it does not.

Applications There are many applications that involve splitting an array/string or cutting a rod. This relates to splitting type of dynamic programming.

5.5 Merge

5.6 More Combinatorics

Combinatorics is about enumerating specified structures, there are some structures are of our main interests through this book and often appears in the interviews, they are: *subarray*, *subsequence*, and *subsets*.

Subarray We have solved one example with subarray. Subarray is defined as a contiguous sequence in the array, which can be represented as $a[i, \dots, j]$. The number of subarray exist in an array of size n will be:

$$sa = \sum_{i=1}^{i=n} i = n * (n + 1) / 2 \quad (5.14)$$

A substring is a contiguous sequence of characters within a string. For instance, "the best of" is a substring of "It was the best of times". This is not to be confused with subsequence, which is a generalization of substring. For example, "Itwastimes" is a subsequence of "It was the best of times", but not a substring.

Prefix and suffix are special cases of substring. A prefix of a string S is a substring of S that occurs at the beginning of S . A suffix of a string S is a substring that occurs at the end of S .

Subsequence For a subsequence means any sequence we can find the array, which is not required to be contiguous, but the ordering still matters. For example, in the array of [ABCD], the subsequence will be

```

1   []
2   [A] ,      [B] ,      [C] ,      [D] ,
3   [AB] , [AC] , [AD] ,      [BC] , [BD] ,      [CD] ,
4   [ABC] , [ABD] , [ACD] , [BCD] ,
5   [ABCD]

```

You would actually see for $n = 4$, there are 16 possible subsequence, which is 2^4 . This is not coincidence. Imagine for each item in the array, they have two options, either be chosen into the possible sequence or not chosen, which make it to 2^n .

$$ss = 2^n \quad (5.15)$$

Subset The Subset B of a set A is defined as a set within all elements of this subset are from set A . In other words, the subset B is contained inside the set A , $B \in A$. There are two kinds of subsets: if the order of the subset does't matter, it is a combination problem, otherwise, it is a permutation problem.

If it is the case that ordering does not matter, for n distinct things, the number of possible subsets, also called *the power set* will be:

$$\text{powerset} = C(n, 0) + C(n, 1) + \dots + C(n, n) \quad (5.16)$$

6

Recurrence Relations

As we mentioned briefly about the power of recursion is in the whole algorithm design and analysis, we dedicate this chapter to recurrence relation. To summarize, recurrence relation can help with:

- Recurrence relation naturally represent the relation of recursion. Examples will be shown in Chapter. 13.
- Any iteration can be translated into recurrence relation. Some examples can be found in Chapter. IV.
- Recurrence relation together with mathematical induction is the most powerful tool to **design** and **prove** the correctness of algorithm(chapter. 13 and Chapter. IV).
- Recurrence relation can be applied to algorithm complexity analysis(Chapter. IV).

In the following chapters of this part, we endow application meanings to these formulas and discuss how to realize the mentioned uses.

6.1 Introduction

Definition and Concepts A recurrence relation is function expressed with the same function. More precisely, as defined in mathematics, recurrence relation is an equation that recursively defines a sequence or multi-dimensional array of values; once one or more initial terms are given, each further term of the sequence or array is defined as a function of the preceding terms. Fibonacci sequence is one of the most famous recurrence relation

which is defined as $f(n) = f(n - 1) + f(n - 2)$, $f(0) = 0$, $f(1) = 1$.

$$a_n = \Psi(n, a_{n-1}) \text{ for } n \leq 0, \quad (6.1)$$

We use a_n to denote the value at index n , and the recurrence function is marked as $\Psi(n, P)$, P is all preceding terms that needed to build up this recurrence relation. Like the case of factorial, each factorial number only relies on the result of the previous number and its current index, this recurrence relation can be written as the following equation:

A recurrence relation needs to start from *initial value(s)*. For the above relation, a_0 needs to be defined and it will be the first element of a recurrence relation. The above relation is only related to the very first preceding terms, which is called recurrence relation of *first order*. If P includes multiple preceding terms, a recurrence relation of order k can be easily extended as:

$$a_n = \Psi(n, a_{n-1}, a_{n-2}, \dots, a_{n-k}) \text{ for } n \leq k, \quad (6.2)$$

In this case, k initial values are needed for defining a sequence. Initial values can be given any values but then once initial values are decided, the recurrence determines the sequence uniquely. Thus, initial values are also called the *degree of freedom* for solutions to the recurrence.

Many natural functions are easily expressed as recurrence:

- Polynomial: $a_n = a_{n-1} + 1, a_1 = 1 \rightarrow a_n = n$.
- Exponential: $a_n = 2 \times a_{n-1}, a_1 = 1 \rightarrow a_n = 2^{n-1}$.
- Factorial: $a_n = n \times a_{n-1}, a_1 = 1 \rightarrow a_n = n!$

Solving Recurrence Relation In real problems, we might care about the value of recursion at n , that is compute a_n for any given n , and there are two ways to do it:

- Programming: we utilize the computational power of computer and code in either iteration or recursion to build up the value at any given n . For example, $f(2) = f(1) + f(0) = 1$, $f(3) = f(2) + f(1) = 2$, and so on. With this iteration, we would need $n-1$ steps to compute $f(n)$.
- Math: we solve the recurrence relation by obtaining an explicit or closed-form expression which is a non-recursive function of n . With the solution at hand, we can get a_n right away.

Recurrence relations plays an important role in the analysis of algorithms. Usually, time recurrence relation $T(n)$ is defined to analyze the time complexity of solving a problem with input instance of size n . The field of complexity analysis studies the closed-form solution of $T(n)$; that is to say

the functional relation between $T(n)$ with n that it cares, not each exact value.

In this section, we focus on solving the recurrence relation using math to get a closed-form solution. Categorizing the recurrence relation can help us pinpoint each type's solving methods.

Categorizes Recurrence relation is essentially discrete function, which can be naturally categorized as **linear** (such as function $y = mx + b$) and **non-linear**; quadratic, cubic and so on (such as $y = ax^2 + bx + c, y = ax^3 + bx^2 + cx + d$). In the field of algorithmic problem solving, linear recurrence relation is commonly used and researched, thus we deliberately leave the non-linear recurrence relation and its method of solving out of the scope of this book.

- **Homogeneous linear recurrence relation:** When the recurrent relation is linear homogeneous of degree k with constant coefficients, it is in the form, and is also called order- k homogeneous linear recurrence with constant coefficients.

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}. \quad (6.3)$$

a_0, a_1, \dots, a_{k-1} will be initial values.

- **Non-homogeneous linear recurrence relation:** An order- k non-homogeneous linear recurrence with constant coefficients is defined in the form:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k} + f(n). \quad (6.4)$$

$f(n)$ can be 1 or n or n^2 and so on.

- **Divide-and-conquer recurrence relation:** When n is not decreasing by a constant as does in Eq. 6.3 and Eq. 6.4, instead by a constant factor, with the equality as shown below, it is called divide and conquer recurrence relation.

$$a_n = a_{n/b} + f(n) \quad (6.5)$$

where $a \leq 1, b > 1$, and $f(n)$ is a given function, which usually has $f(n) = cn^k$.

We will introduce general methods to solve a linear recurrence relation but leave out the part of divide and conquer recurrence relation in this chapter for reason that divide and conquer recurrence relation will most likely to be solved with just roughly, as shown in Chapter. IV to just estimate the time complexity resulted from the divide and conquer method.

6.2 General Methods to Solve Linear Recurrence Relation

No general method for solving recurrence function is known yet, however, linear recurrence relation with finite initial values and previous states, constant coefficients can always be solved. Due to the fact that the recursion is essentially mathematical induction, the most general way of solving any recurrence relation is to use *mathematical induction* and *iterative method*. This also makes the the mathematical induction, in some form, the foundation of all correctness proofs for computer programs. We examine these two methods by solving two recurrence relation: $a_n = 2 \times a_{n-1} + 1, a_0 = 0$ and $a_n = a_{n/2} + 1$.

6.2.1 Iterative Method

The most straightforward method for solving recurrence relation no matter its linear or non-linear is the *iterative method*. Iterative method is a technique or procedure in computational mathematics that it iteratively replace/substitute each a_n with its recurrence relation $\Psi(n, a_{n-1}, a_{n-2}, \dots, a_{n-k})$ till all items “disappear” other than the initial values. Iterative method is also called substitution method.

We demonstrate iteration with a simple non-overlapping recursion.

$$T(n) = T(n/2) + O(1) \quad (6.6)$$

$$= T(n/2^2) + O(1) + O(1)$$

$$= T(n/2^3) + 3O(1)$$

$$= \dots$$

$$= T(1) + kO(1) \quad (6.7)$$

We have $\frac{n}{2^k} = 1$, we solve this equation and will get $k = \log_2 n$. Most likely $T(1) = O(1)$ will be the initial condition, we replace this, and we get $T(n) = O(\log_2 n)$.

However, when we try to apply iteration on the third recursion: $T(n) = 3T(n/4) + O(n)$. It might be tempting to assume that $T(n) = O(n \log n)$ due to the fact that $T(n) = 2T(n/2) + O(n)$ leads to this time complexity.

$$T(n) = 3T(n/4) + O(n) \quad (6.8)$$

$$\begin{aligned} &= 3(3T(n/4^2) + n/4) + n = 3^2T(n/4^2) + n(1 + 3/4) \\ &= 3^2(3T(n/4^3) + n/4^2) + n(1 + 3/4) = 3^3T(n/4^3) + n(1 + 3/4 + 3/4^2) \end{aligned} \quad (6.9)$$

$$= \dots \quad (6.10)$$

$$= 3^kT(n/4^k) + n \sum_{i=0}^{k-1} \left(\frac{3}{4}\right)^i \quad (6.11)$$

6.2.2 Recursion Tree

Since the term of $T(n)$ grows, the iteration can look messy. We can use recursion tree to better visualize the process of iteration. In a recursive tree, each node represents the value of a single subproblem, and a leaf would be a subproblem. As a start, we expand $T(n)$ as a node with value n as root, and it would have three children each represents a subproblem $T(n/4)$. We further do the same with each leaf node, until the subproblem is trivial and be a base case. In practice, we just need to draw a few layers to find the rule. The cost will be the sum of costs of all layers. The process can be seen in Fig. 10.3. In this case, it is the base case $T(1)$. Through the expansion

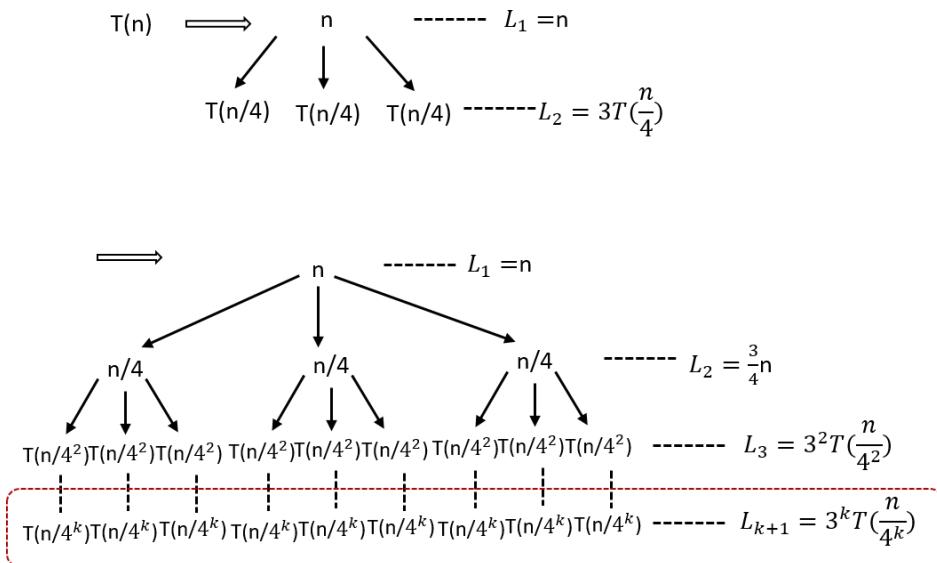


Figure 6.1: The process to construct a recursive tree for $T(n) = 3T(\lfloor n/4 \rfloor) + O(n)$. There are totally $k+1$ levels. Use a better figure.

with iteration and recursion tree, our time complexity function becomes:

$$T(n) = \sum_{i=1}^k L_i + L_{k+1} \quad (6.12)$$

$$= n \sum_{i=1}^k (3/4)^{i-1} + 3^k T(n/4^k) \quad (6.13)$$

In the process, we can see that Eq. 10.13 and Eq. 10.7 are the same.

Because $T(n/4^k) = T(1) = 1$, we have $k = \log_4 n$.

$$T(n) \leq n \sum_{i=1}^{\infty} (3/4)^{k-1} + 3^k T(n/4^k) \quad (6.14)$$

$$\leq 1/(1 - 3/4)n + 3^{\log_4 n} T(1) = 4n + n^{\log_4 3} \leq 5n \quad (6.15)$$

$$= O(n) \quad (6.16)$$

6.2.3 Mathematical Induction

Mathematical induction is a mathematical proof technique, and is essentially used to prove that a property $P(n)$ holds for every natural number n , i.e. for $n = 0, 1, 2, 3$, and so on. Therefore, in order to use induction, we need to make a *guess* of the closed-form solution for a_n . Induction requires two cases to be proved.

1. *Base case*: proves that the property holds for the number 0.
2. *Induction step*: proves that, if the property holds for one natural number n , then it holds for the next natural number $n + 1$.

For $T(n) = 2 \times T(n - 1) + 1$, $T_0 = 0$, we can have the following result by expanding $T(i)$, $i \in [0, 7]$.

n	0	1	2	3	4	5	6	7
T_n	0	3	7	15	31	63	127	

It is not hard that we find the rule and guess $T(n) = 2^n - 1$. Now, we prove this equation by induction:

1. Show that the basis is true: $T(0) = 2^0 - 1 = 0$.
2. Assume it holds true for $T(n - 1)$. By induction, we get

$$T(n) = 2T(n - 1) + 1 \quad (6.17)$$

$$= 2(2^{n-1} - 1) + 1 \quad (6.18)$$

$$= 2^n - 1 \quad (6.19)$$

Now we show that the induction step holds true too.

 Solve $T(n) = T(n/2) + O(1)$ and $T(2n) \leq 2T(n) + 2n - 1$, $T(2) = 1$.

Briefing on Other Methods When the form of the linear recurrence is more complex, say large degree of k , more complex of the $f(n)$, none of the iterative and induction methods is practical and manageable. For iterative method, the expansion will be way too messy for us to handle. On the side of induction method, it is quite challenging or sometimes impossible for us just to “guess” or “generalize” the exact closed-form of recurrence relation solution purely based on observing a range of expansion.

The more general and approachable method for solving homogeneous linear recurrence relation derives from making a rough guess rather than exact guess, and then solve it via *characteristic equation*. This general method is pinpointed in Section. 6.3 with examples. For non-homogeneous linear recurrence relation (Section. 6.4), there are generally two ways – *symbolic differentiation* and *method of undetermined coefficients* to solve non-homogeneous linear recurrence relation and both of them relates to solving homogeneous linear relation. The study of the remaining content is most math saturated in the book, while we later on will find out its tremendous help in complexity analysis in Chapter. IV and potentially in problem solving.

6.3 Solve Homogeneous Linear Recurrence Relation

In this section, we offer a more general and more manageable method for solving recurrence relation that is homogeneous defined in Eq. 6.3. There are three broad methods: using characteristic equation which we will learn in this section, and the other two– linear algebra, and Z-transofrm¹ will not be included.

Make a General “Guess” From our previous examples, we can figure out the closed-form solution for simplified homogeneous linear recurrence such as the fibonacci recurrence relation:

$$a_n = a_{n-1} + a_{n-2}, a_0 = 0, a_1 = 1 \quad (6.20)$$

A reasonable guess would be that a_n is doubled every time; namely, it is approximately 2^n . Let’s guess $a_n = c2^n$ for some constant c . Now we substitute Eq. 6.21, we get

$$c2^n = c2^{n-1} + c2^{n-2} = c2^n \quad (6.21)$$

We can see that c will be canceled and the left side is always greater than the right side. Thus we learned that $c2^n$ is a too large guess, and the multiplicative constant c plays no role in the induction step.

¹Visit https://en.wikipedia.org/wiki/Recurrence_relation for details.

Based on the above example, we introduce a parameter γ as a base, $a_n = \gamma^n$ for some γ . We then compute its value through solving *Characteristic Equation* as introduced below.

Characteristic Equation Now, we substitute our guess into the Eq.6.3, then

$$\gamma^n = a_n \quad (6.22)$$

$$= c_1\gamma^{n-1} + c_2\gamma^{n-2} + \dots + c_k\gamma^{n-k}. \quad (6.23)$$

We rewrite Eq. 6.23 as:

$$\gamma^n - c_1\gamma^{n-1} - c_2\gamma^{n-2} - \dots - c_k\gamma^{n-k} = 0. \quad (6.24)$$

By dividing γ^{n-k} from left and right side of the equation, we get the simplified equation, which is called the *characteristic equation* of the recurrence relation in the form of Eq. 6.3.

$$\gamma^k - c_1\gamma^{k-1} - c_2\gamma^{k-2} - \dots - c_k = 0. \quad (6.25)$$

The concept of characteristic equation is related to generating function². The solutons of characteristic equation are called *characteristic roots*.

Characteristic Roots and Solution Now, we have a linear homogeneous recurrence relation and its characteristic equation, and assume that the equation has k distinct roots, $\gamma_1, \gamma_2, \dots, \gamma_k$, then we can build upon these chracteristic roots, the general guess, and some other k constants, d_1, d_2, \dots, d_k of $\{a_n\}$ as:

$$a_n = d_1\gamma_1^n + d_2\gamma_2^n + \dots + d_k\gamma_k^n \quad (6.26)$$

The unknown constants, d_1, d_2, \dots, d_k of $\{a_n\}$ can be found using the initial values a_0, a_1, \dots, a_{k-1} by solving the following equations:

$$a_0 = d_1\gamma_1^0 + d_2\gamma_2^0 + \dots + d_k\gamma_k^0, \quad (6.27)$$

$$a_1 = d_1\gamma_1^1 + d_2\gamma_2^1 + \dots + d_k\gamma_k^1, \quad (6.28)$$

$$\dots, \quad (6.29)$$

$$a_{k-1} = d_1\gamma_1^{k-1} + d_2\gamma_2^{k-1} + \dots + d_k\gamma_k^{k-1}. \quad (6.30)$$

Within the context of computer science, the degree is mostly within 2. Here, we introduce the formula solving the character roots for characteristic equation with the following form:

$$0 = ax^2 + bx + c \quad (6.31)$$

²

6.4. SOLVE NON-HOMOGENEOUS LINEAR RECURRENCE RELATION83

The root(s) can be computed from the following formula ³ :

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (6.32)$$

Hands-on Example For $a_n = 2a_{n-1} + 3a_{n-2}$, $a_0 = 3$, $a_1 = 5$, we can write the characteristic equation as $\gamma^2 - 2\gamma - 3 = 0$. Because $\gamma^2 - 2\gamma - 3 = (\gamma - 3) + (\gamma + 1)$, which make the characteristic roots $\gamma_1 = 3$, $\gamma_2 = -1$. Now our solution has the form:

$$a_n = d_1 3^n + d_2 (-1)^n \quad (6.33)$$

Now, we find the constants via listing the initial values we know:

$$a_0 = d_1 3^0 + d_2 (-1)^0 = d_1 + d_2 = 3, \quad (6.34)$$

$$a_1 = d_1 3^1 + d_2 (-1)^1 = 3d_1 - d_2 = 5. \quad (6.35)$$

We would get $d_1 = 2$, $d_2 = 1$. Finally, we have a solution $a_n = 2*3^n + (-1)^n$.



Continue to solve $a_n = a_{n-1} + a_{n-2}$.

6.4 Solve Non-homogeneous Linear Recurrence Relation

method of undetermined coefficients where the solution is comprised of the solution of the homogeneous part and the particular $f(n)$ part by summing up; and the method of *symbolic differentiation* which converts from the equation the same form of homogeneous linear recurrence relation.

The complexity analysis for most algorithms fall into the form of non-homogeneous linear recurrence relation. For examples: in fibonacci sequence, if it is solved by using recursion shown in Chapter. 15 without caching mechanism, the time recurrence relation is $T(n) = T(n-1) + T(n-2) + 1$; in the merge sort discussed in Chapter. 13, the recurrence relation is $T(n) = T(n/2) + n$. Examples of recurrence relation $T(n) = T(n-1) + n$ can be easily found, such as the maximum subarray.

Method of Undetermined Coefficients Suppose we have a recurrence relation in the form of Eq. 6.4.

Suppose we ignore the non-linear part and just look at the homogeneous part:

$$h_n = c_1 h_{n-1} + c_2 h_{n-2} + \dots + c_k h_{n-k}. \quad (6.36)$$

³Visit <http://www.biology.arizona.edu/biomath/tutorials/Quadratic/Roots.html> for derivation

Symbolic Differentiation

6.5 Useful Math Formulas

Knowing these facts can be very important in practice, we can treat each as an element in the problem solving. Sometimes, when its hard to get the closed form of a recurrence relation or finding the recurrence relation, we decompose it to multiple parts with these elements. Put some examples.

binomial theorem

$$\sum_{k=0}^n C_n^k x^k = (1 + x)^n \quad (6.37)$$

An example of using this the cost of generating a powerset, where $x = 1$.

6.6 Exercises

1. Compute factorial sequence using `while` loop.
2. Greatest common divisor: The Euclidean algorithm, which computes the greatest common divisor of two integers, can be written recursively.

$$gcd(x, y) = \begin{cases} x & \text{if } y = 0, \\ gcd(y, x \% y) & \text{if } y > 0 \end{cases} \quad (6.38)$$

Function definition:

6.7 Summary

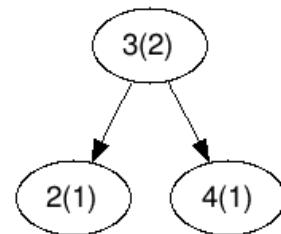
If a cursive algorithm can be further optimized, the optimization method can either be divide and conquer or decrease and conquer. We have put much effort into solving recurrence relation of both: the linear recurrence relation for decrease and conquer, the divide and conquer recurrence relation for divide and conquer. Right now, do not struggle and eager to know what is divide or decrease and conquer, it will be explained in the next two chapters.

Further, Akra-Bazzi Method ⁴ applies to recurrence such that $T(n) = T(n/3) + T(2n/3) + O(n)$. Please look into more details if interested. Generating function is used to solve the linear recurrence.

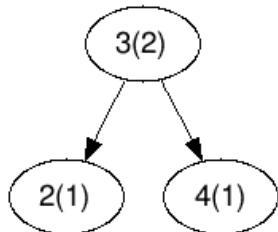
Part III

Get Started: Programming and Python Data Structures

After the warm up, we prepare ourselves with hands-on skills—basic programming with Python 3, including two function type—iteration and recursion, and connecting dots between the abstract data structures with Python



3 built-in data types and commonly used modules.



Python is object-oriented programming language and its underlying implementation is C++, which has a good mapping with the abstract data structures we discussed. Learn how to use Python data type can be learned from the official Python tutorial: <https://docs.python.org/3/tutorial/>. However, in order to grasp the efficiency of data structures needs us to examine its C++ source code (<https://github.com/python/cpython>) that relates easily to abstract data structures.

7

Iteration and Recursion

“The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.”

– Niklaus Wirth, *Algorithms + Data Structures = Programs*, 1976

7.1 Introduction

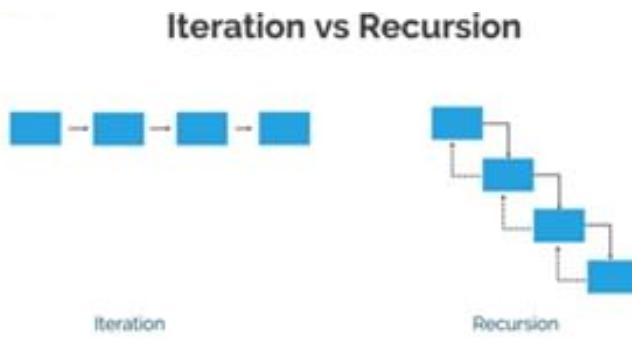


Figure 7.1: Iteration vs recursion: in recursion, the line denotes the top-down process and the dashed line is the bottom-up process.

In computer science, software programs can be categorized as either *iteration* or *recursion*, thus making iteration and recursion as the topmost level of concepts in software development and the very first base for us to study

computer science techniques. Iteration refers to a *looping* process which repeats some part of the code until a certain condition is met. Recursion, similarly, needs to stop at a certain condition, but it replaces the loop with recursive function calls; meaning a function calls itself from within its own code. The process is shown in Fig. 7.1.

Do you still have the feeling that you seemingly already understand the iteration even without code, but what is recursion exactly? Recursion can be a bit of challenging for beginners, it differs from our normal way of thinking. It is a bit of similar to the vision of being in the restroom which has two mirrors abreast on each side and facing each other, we see multiple images of the things in front of each mirror, and these images usually appear from large to small. This is similar to recursion. The relation between these recurred images can be called *recurrence relation*.

Understanding recursion and learning basic rules to solve recurrence relation are two of the most purposes in this chapter. Thus, we organize the content of this chapter following this trend:

1. Section. ?? will first address our question by analyzing the recursion mechanism within the computer program, and we further understand the different between by seeing example of factorial series and examines the pros and cons of each.
2. Section. 7.4 advances our knowledge about recursion by studying the recurrence relation, including its definition, categorization and addressing how to solve recurrence relation.
3. Section. ?? gives us two examples to see how iteration and recursion works in real practice.



Deduce/find) the recurrence relation and sometimes solves it is a key step in algorithm design and problem solving, solving the recurrence time relation is important to algorithm analysis.

In this section, we first learn iteration and Python Syntax that can be used to implement. We then examine a classic and elementary example—Factorial sequence to catch a glimpse of how iteration and recursion can be applied to solve this problem. Then, we discuss more details about recursion. We end this section by comparing iteration and recursion; their pros and cons and their relation between.

7.2 Iteration

In simple terms, an iterative function is one that loops to repeat some part of the code. In Python, the loops can be expressed with `for` and `while`

loop.

Enumerating the number from 1 to 10 is a simple iteration. Implementation wise:

- **for** usually is used together with function `range(start, stop, step)` which creates a sequence of numbers from `start` to `stop` in range $[start, end]$, and increments by `step` (1 by default). Thus, we need to set `start` as 1, and `end` as 11 to get numbers from 1 to 10.

```
1 # enumerate 1 to 10 with for loop
2 for i in range(1, 11):
3     print(i, end=' ')
```

- **while** is used with syntax

```
while expression
    statement
```

In our case, we need to set start condition which is $i = 1$, and the expression will be limiting $i \leq 10$. In the statement, we need to manually increment the variable `i` so that we wont not end up with infinite loop.

```
1 i = 1
2 while i <= 10:
3     print(i, end = ' ')
4     i += 1
```

7.3 Factorial Sequence

The factorial of a positive integer n , denoted by $n!$, is the product of all positive integers less than or equal to n :

```
For example:
5! = 5 \times 4 \times 3 \times 2 \times 1 = 120.
0! = 1
```

To compute the factorial sequence at n , we need to know the factorial sequence at $n - 1$, which can be expressed as a *recurrence relation*, that $n! = n \times (n - 1)!$.

- Solving with iteration: we use a `for` loop starts at 1 up till n so that we eventually build up our answer at n . We use a variable `ans` to save the factorial result for each number, and once the program stops, `ans` gives the result of our factorial for n .

```
1 def factorial_iterative(n):
2     ans = 1
3     for i in range(1, n+1):
4         ans = ans * i
5     return ans
```

- Solving with recursion: we start to call a recursive function at n , within this function, we can itself but instead with $n - 1$ just as shown in the recurrence relation. We then multiply this recursive call with n . We need to define a bottom, which is the end condition for the recursive function calls to avoid infinite loop. In this case, it bottoms out at $n = 1$, which we can know its answer would be 1, thus we return 1 to stop further function calls and recursively return to its upmost level.

```

1 def factorial_recursive(n):
2     if n == 1:
3         return 1
4     return n * factorial_recursive(n-1)

```

7.4 Recursion

In this section, we reveal how the recursion mechanism works: function calls and stack, two passes.

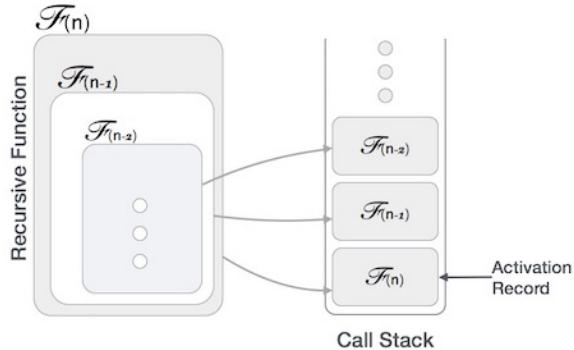


Figure 7.2: Call stack of recursion function

Two Elements When a routine calls itself either directly or indirectly, it is said to be making a recursive function call. The basic idea behind solving problems via recursion is to break the instance of the problem into smaller and smaller instances until the instances are so small they can be solved trivially. We can view a recursive routine as consisting of two parts.

- Recursive Calls: As in the factorial sequence, when the instance of the problem is still too large to solve directly, we recursive call this function itself to solve problems of smaller size. Then the result returned from the recursive calls are used to build upon the result of the upper level using *recurrence relation*. For example, If we use $f(n)$ to denote the factorial at n , the recurrence relation would be $f(n) = n \times f(n-1)$, $n > 0$.

- End/Basis Cases: The above recursive call needs to bottom-out; stop when the instance is so small to be solved directly. This stop condition is called end/basic case. Without this case, the recursion will continue to dive infinitely deep and eventually we run out of memory and get a crash. A recursive function can have one or more base cases. In the example of factorial, the base case is when $n = 0$, by definition $0! = 1$.

Recursive Calls and Stacks The recursive function calls of the recursive factorial we implemented in the last section can be demonstrated as Fig. 7.2.

The execution of recursive function $f(n)$ will pay two visits to each recursive function $f(i), i \in [1, n]$ through two passes: *top-down* and *bottom-up* as we have illustrated in Fig. 7.1. The recursive function handles this process via a *stack* data structure which follows a Last In First Out (LIFO) principle to record all function calls.

- In the top-down pass, each recursive function's execution context is “pushed” into the stack in the order of $f(n), f(n - 1), \dots, f(1)$. The process ends till it hits to the end case $f(0)$, which will not be “pushed” into the stack but executes some code and **returns** value(s). The end case marks as the start of the bottom-up process.
- In the bottom-up pass, the recursive function's execution context in the stack is “popped” off the stack in a reversed order: $f(1), \dots, f(n - 1), f(n)$. And $f(1)$ takes the returned value from the base case to construct its value using the recurrence relation. Then it returns its value up to the next recursive function $f(2)$. This whole process ends at $f(n)$ which returns its value.

How Important Recursion Is? Recursion is a very powerful and fundamental technique, and it is basis for several other design principles, such as:

- Divide and Conquer (Chapter. 13).
- Recursive Search, such as Tree Traversal and graph search.
- Dynamic Programming (Chapter. 15).
- Combinatorics such as enumeration (permutation and combination) and branch and bound etc.
- Some classes of greedy algorithms.

It also supports the proof of correctness of algorithms via mathematical induction, and consistently arise in the algorithm complexity analysis. We shall see through out this book and will end up drawing this conclusion ourselves.

Practical Guideline In real algorithmic problem solving, different process normally has different usage.

In top-down process we do:

1. Break problems into smaller problems, there are different ways of “breaking” and depends on which, they can either be *divide and conquer* or *decrease and conquer* which we will further expand in Chapter. ?? and ?. Divide and conquer will divide the problems into disjoint subproblems, whereas in decrease and conquer, the problems
2. Searching: visit nodes in non-linear data structures (graph/tree), visit nodes in linear data structures. Also, at the same time, we can use **pass by reference** to track the state change such as the traveled path in the path related graph algorithms.

In bottom-up process, we can either return `None` or `variables`. Assume if we already used **pass by reference** to tack the change of state, then it is not necessarily to return variables. In some scenario, tracking states with by passing by reference can be more easier and more intuitive. For example, in the graph algorithm, we mostly like to use this method.

Tail Recursion This is also called *tail recursion* where the function calls itself at the end (“tail”) of the function in which no computation is done after the return of recursive call. Many compilers optimize to change a recursive call to a tail recursive or an iterative call.

7.5 Iteration VS Recursion

Stack Overflow Problem In our example, if we call function `factorial_recursive()` with $n = 1000$, Python would have complain an error as:

```
1 RecursionError: maximum recursion depth exceeded in comparison
```

which is a *stack overflow* problem. A stack overflow is when we run out of memory to hold items in the stack. These situations can incur the stack overflow problem:

1. No base case is defined.
2. The recursion is too deep which is out of the assigned memory limit of the executing machine.

Stack Overflow for Recursive Function and Iterative Implementation According to Wikipedia, in software, a stack overflow occurs if the call stack pointer exceeds the stack bound. The call stack may consist of a limited amount of address space, often determined at the start of the program

depending on many factors, including the programming language, machine architecture, multi-threading, and amount of available memory. When a program attempts to use more space than is available on the call stack, the stack is said to *overflow*, typically resulting in a program crash. The very deep recursive function is faced with the threat of stack overflow. And the only way we can fix it is by transforming the recursion into a loop and storing the function arguments in an explicit stack data structure, this is often called the iterative implementation which corresponds to the recursive implementation.

We need to follow these points:

1. End condition, Base Cases and Return Values: either return an answer for base cases or None, and used to end the recursive calls.
2. Parameters: parameters include: data needed to implement the function, current paths, the global answers and so on.
3. Variables: What the **local** and global variables. In Python any pointer type of data can be used as global variable global result putting in the parameters.
4. Construct current result: when to collect the results from subtree and combine to get the result for current node.
5. Check the depth: if the program will lead to the heap stack overflow.

Conversion For a given problem, conversion between iteration and recursion is possible, but the difficulty of the conversion is highly dependable on specific problem context. For example, the iteration of a range of numbers can be represented with recurrence relation $T(n) = T(n - 1) + 1$. On the side of implementation, some recursion and iteration can be easily converted between such as linear search; in some other cases, it takes more tricks and requires more sophisticated data structures to assist the conversion, such as in the iterative implementation of the recursive depth-first-search, it uses stack. Do not worry about these concepts here, as you flip more pages in the book, you will know and start to think better.

Tail recursion and Optimization In a typical recursive function, we usually make the recursive calls first, and then take the return value of the recursive call to calculate the result. Therefore, we only get the final result after all the recursive calls have returned some value. But in a tail recursive function, the various calculations and statements are performed first and the recursive call to the function is made after that. By doing this, we pass the results of the current step to the next recursive call to the function. Hence, the last statement in a Tail recursive function is the recursive call to the

function. This means that when we perform the next recursive call to the function, the current stack frame (occupied by the current function call) is not needed anymore. This allows us to optimize the code. We simply reuse the current stack frame for the next recursive step and repeat this process for all the other function calls.

Using regular recursion, each recursive call pushes another entry onto the call stack. When the functions return, they are popped from the stack. In the case of tail recursion, we can optimize it so that only one stack entry is used for all the recursive calls of the function. This means that even on large inputs, there can be no stack overflow. This is called Tail recursion optimization.

Languages such as lisp and c/c++ have this sort of optimization. But, the Python interpreter doesn't perform tail recursion optimization. Due to this, the recursion limit of python is usually set to a small value (approx, 10^4). This means that when you provide a large input to the recursive function, you will get an error. This is done to avoid a stack overflow. The Python interpreter limits the recursion limit so that infinite recursions are avoided.

Handling recursion limit The “sys module” in Python provides a function called `setrecursionlimit()` to modify the recursion limit in Python. It takes one parameter, the value of the new recursion limit. By default, this value is usually 10^4 . If you are dealing with large inputs, you can set it to, 10^6 so that large inputs can be handled without any errors.

7.6 Exercises

1. Compute factorial sequence using `while` loop.

7.7 Summary

If a cursive algorithm can be further optimized, the optimization method can either be divide and conquer or decrease and conquer. We have put much effort into solving recurrence relation of both: the linear recurrence relation for decrease and conquer, the divide and conquer recurrence relation for divide and conquer. Right now, do not struggle and eager to know what is divide or decrease and conquer, it will be explained in the next two chapters.

8

Bit Manipulation

Many books on algorithmic problem solving seems forget about one topic—bit and bit manipulation. Bit is how data is represented and saved on the hardware. Thus knowing such concept and bit manipulation using Python sometimes can also help us device more efficient algorithms, either space or time complexity in the later Chapter.

For example, how to convert a char or integer to bit, how to get each bit, set each bit, and clear each bit. Also, some more advanced bit manipulation operations. After this, we will see some examples to show how to apply bit manipulation in real-life problems.

8.1 Python Bitwise Operators

Bitwise operators include `«`, `»`, `&`, `|`, `^`. All of these operators operate on signed or unsigned numbers, but instead of treating that number as if it were a single value, they treat it as if it were a string of bits. Twos-complement binary is used for representing the singed number.

Now, we introduce the six bitwise operators.

x « y Returns x with the bits shifted to the left by y places (and new bits on the right-hand-side are zeros). This is the same as multiplying x by 2^y .

x » y Returns x with the bits shifted to the right by y places. This is the same as dividing x by 2^y , same result as the `//` operator. This right shift is also called *arithmetic right shift*, it fills in the new bits with the value of the sign bit.

x & y "Bitwise and". Each bit of the output is 1 if the corresponding bit of x AND of y is 1, otherwise it's 0. It has the following property:

```

1 # keep 1 or 0 the same as original
2 1 & 1 = 1
3 0 & 1 = 0
4 # set to 0 with & 0
5 1 & 0 = 0
6 0 & 0 = 0

```

x | y "Bitwise or". Each bit of the output is 0 if the corresponding bit of x AND of y is 0, otherwise it's 1.

```

1 # set to 1 with | 1
2 1 | 1 = 1
3 0 | 1 = 1
4
5 # keep 1 or 0 the same as original
6 1 | 0 = 1
7 0 | 0 = 0

```

$\sim x$ Returns the complement of x - the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as $-x - 1$ (really?).

x ^ y "Bitwise exclusive or". Each bit of the output is the same as the corresponding bit in x if that bit in y is 0, and it's the complement of the bit in x if that bit in y is 1. It has the following basic properties:

```

1 # toggle 1 or 0 with ^ 1
2 1 ^ 1 = 0
3 0 ^ 1 = 1
4
5 # keep 1 or 0 with ^ 0
6 1 ^ 0 = 1
7 0 ^ 0 = 0

```

Some examples shown:

```

1 A = 5 = 0101, B = 3 = 0011
2 A ^ B = 0101 ^ 0011 = 0110 = 6
3

```

More advanced properties of XOR operator include:

```

1 a ^ b = c
2 c ^ b = a
3
4 n ^ n = 0
5 n ^ 0 = n
6 eg. a=00111011, b=10100000 , c= 10011011, c ^ b= a
7

```

Logical right shift The logical right shift is different to the above right shift after shifting it puts a 0 in the most significant bit. It is indicated with a `>>>` operator in Java. However, in Python, there is no such operator, but we can implement one easily using `bitstring` module padding with zeros using `>>=` operator.

```

1 >>> a = BitArray(int=-1000, length=32)
2 >>> a.int
3 -1000
4 >>> a >>= 3
5 >>> a.int
6 536870787

```

8.2 Python Built-in Functions

bin() The `bin()` method takes a single parameter `num`- an integer and return its *binary string*. If not an integer, it raises a `TypeError` exception.

```

1 a = bin(88)
2 print(a)
3 # output
4 # 0b1011000

```

However, `bin()` doesn't return *binary bits* that applies the two's complement rule. For example, for the negative value:

```

1 a1 = bin(-88)
2 # output
3 # -0b1011000

```

int(x, base = 10) The `int()` method takes either a string `x` to return an integer with its corresponding base. The common base are: 2, 10, 16 (hex).

```

1 b = int('01011000', 2)
2 c = int('88', 10)
3 print(b, c)
4 # output
5 # 88 88

```

chr() The `chr()` method takes a single parameter of integer and return a character (a string) whose Unicode code point is the integer. If the integer `i` is outside the range, `ValueError` will be raised.

```

1 d = chr(88)
2 print(d)
3 # output
4 # X

```

Decimal value	Binary (two's-complement representation)	Two's complement \Leftrightarrow $(2^8 - n)_2$
0	0000 0000	0000 0000
1	0000 0001	1111 1111
2	0000 0010	1111 1110
126	0111 1110	1000 0010
127	0111 1111	1000 0001
-128	1000 0000	1000 0000
-127	1000 0001	0111 1111
-126	1000 0010	0111 1110
-2	1111 1110	0000 0010
-1	1111 1111	0000 0001

Figure 8.1: Two's Complement Binary for Eight-bit Signed Integers.

ord() The `ord()` method takes a string representing one Unicode character and return an integer representing the Unicode code point of that character.

```

1 e = ord('a')
2 print(e)
3 # output
4 # 97

```

8.3 Twos-complement Binary

Given 8 bits, if it is unsigned, it can represent the values 0 to 255 (1111,1111). However, a two's complement 8-bit number can only represent positive integers from 0 to 127 (0111,1111) because the most significant bit is used as sign bit: '0' for positive, and '1' for negative.

$$\sum_{i=0}^{N-1} 2^i = 2^{(N-1)} + 2^{(N-2)} + \dots + 2^2 + 2^1 + 2^0 = 2^N - 1 \quad (8.1)$$

The twos-complement binary is the same as the classical binary representation for positive integers and differs slightly for negative integers. Negative integers are represented by performing Two's complement operation on its absolute value: it would be $(2^N - n)$ for representing $-n$ with N-bits. Here, we show Two's complement binary for eight-bit signed integers in Fig. 8.1.

Get Two's Complement Binary Representation In Python, to get the two's complement binary representation of a given integer, we do not really have a built-in function to do it directly for negative number. Therefore, if we want to know how the two's complement binary look like for negative integer we need to write code ourselves. The Python code is given as:

```

1 bits = 8
2 ans = (1 << bits) - 2
3 print(ans)
4 # output
5 # '0b11111110'
```

There is another method to compute: inverting the bits of n (this is called **One's Complement**) and adding 1. For instance, use 8 bits integer 5, we compute it as the follows:

$$5_{10} = 0000,0101_2, \quad (8.2)$$

$$-5_{10} = 1111,1010_2 + 1_2, \quad (8.3)$$

$$-5_{10} = 1111,1011_2 \quad (8.4)$$

To flip a binary representation, we need expression $x \text{ XOR } '1111,1111'$, which is $2^N - 1$. The Python Code is given:

```

1 def twoes_complement(val , bits):
2     # first flip implemented with xor of val with all 1's
3     flip_val = val ^ (1 << bits - 1)
4     #flip_val = ~val we only give 3 bits
5     return bin(flip_val + 1)
```

Get Two's Complement Binary Result In Python, if we do not want to see its binary representation but just the result of two's complement of a given positive or negative integer, we can use two operations $-x$ or $\sim +1$. For input 2, the output just be a negative integer -2 instead of its binary representation:

```

1 def twoes_complement_result(x):
2     ans1 = -x
3     ans2 = ~x + 1
4     print(ans1 , ans2)
5     print(bin(ans1) , bin(ans2))
6     return ans1
7 # output
8 # -8 -8
9 # -0b1000 -0b1000
```

This is helpful if we just need two's complement result instead of getting the binary representation.

8.4 Useful Combined Bit Operations

For operations that handle each bit, we first need a *mask* that only set that bit to 1 and all the others to 0, this can be implemented with arithmetic left shift sign by shifting 1 with 0 to n-1 steps for n bits:

```
1 mask = 1 << i
```

Get ith Bit In order to do this, we use the property of AND operator either 0 or 1 and with 1, the output is the same as original, while if it is and with 0, they others are set with 0s.

```
1 # for n bit , i in range [0 ,n-1]
2 def get_bit(x, i):
3     mask = 1 << i
4     if x & mask:
5         return 1
6     return 0
7 print(get_bit(5,1))
8 # output
9 # 0
```

Else, we can use left shift by i on x, and use AND with a single 1.

```
1 def get_bit2(x, i):
2     return x >> i & 1
3 print(get_bit2(5,1))
4 # output
5 # 0
```

Set ith Bit We either need to set it to 1 or 0. To set this bit to 1, we need matching relation: $1 -> 1, 0 -> 1$. Therefore, we use operator `|`. To set it to 0: $1 -> 0, 0 -> 0$. Because $0 \& 0/1 = 0, 1\&0=1, 1\&1 = 1$, so we need first set that bit to 0, and others to 1.

```
1 # set it to 1
2 x = x | mask
3
4 # set it to 0
5 x = x & (~mask)
```

Toggle ith Bit Toggling means to turn bit to 1 if it was 0 and to turn it to 0 if it was one. We will be using 'XOR' operator here due to its properties.

```
1 x = x ^ mask
```

Clear Bits In some cases, we need to clear a range of bits and set them to 0, our base mask need to put 1s at all those positions, Before we solve this problem, we need to know a property of binary subtraction. Check if you can find out the property in the examples below,

```
1000-0001 = 0111
0100-0001 = 0011
1100-0001 = 1011
```

The property is, the difference between a binary number n and 1 is all the bits on the right of the rightmost 1 are flipped including the rightmost 1. Using this amazing property, we can create our mask as:

```
1 # base mask
2 i = 5
3 mask = 1 << i
4 mask = mask -1
5 print(bin(mask))
6 # output
7 # 0b11111
```

With this base mask, we can clear bits: (1) All bits from the most significant bit till i (leftmost till i th bit) by using the above mask. (2) All bits from the least significant bit to the i th bit by using $\sim mask$ as mask. The Python code is as follows:

```
1 # i i-1 i-2 ... 2 1 0, keep these positions
2 def clear_bits_left_right(val, i):
3     print('val', bin(val))
4     mask = (1 << i) -1
5     print('mask', bin(mask))
6     return bin(val & (mask))

1 # i i-1 i-2 ... 2 1 0, erase these positions
2 def clear_bits_right_left(val, i):
3     print('val', bin(val))
4     mask = (1 << i) -1
5     print('mask', bin(~mask))
6     return bin(val & (~mask))
```

Run one example:

```
print(clear_bits_left_right(int('11111111',2), 5))
print(clear_bits_right_left(int('11111111',2), 5))
val 0b11111111
mask 0b11111
0b11111
val 0b11111111
mask -0b100000
0b11100000
```

Get the lowest set bit Suppose we are given '0010,1100', we need to get the lowest set bit and return '0000,0100'. And for 1100, we get 0100. If we try to do an AND between 5 and its two's complement as shown in Eq. 8.2 and 8.4, we would see only the right most 1 bit is kept and all the others are cleared to 0. This can be done using expression $x \& (-x)$, $-x$ is the two's complement of x .

```

1 def get_lowest_set_bit(val):
2     return bin(val & (-val))
3 print(get_lowest_set_bit(5))
4 # output
5 # 0b1

```

Or, optionally we can use the property of subtracting by 1.

```

1 x ^ (x & (x - 1))

```

Clear the lowest set bit In many situations we want to strip off the lowest set bit for example in Binary Indexed tree data structure, counting number of set bit in a number. We use the following operations:

```

1 def strip_last_set_bit(val):
2     print(bin(val))
3     return bin(val & (val - 1))
4 print(strip_last_set_bit(5))
5 # output
6 # 0b101
7 # 0b100

```

8.5 Applications

Recording States Some algorithms like Combination, Permutation, Graph Traversal require us to record states of the input array. Instead of using an array of the same size, we can use a single integer, each bit's location indicates the state of one element with same index in the array. For example, we want to record the state of an array with length 8. We can do it like follows:

```

1 used = 0
2 for i in range(8):
3     if used &(1<<i): # check state at i
4         continue
5     used = used | (1<<i) # set state at i used
6     print(bin(used))

```

It has the following output

```

0b1
0b11
0b111
0b1111
0b11111
0b111111
0b1111111
0b11111111

```

XOR Single Number

8.1 **136. Single Number(easy).** Given a non-empty array of integers, every element appears twice except for one. Find that single one.

Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Example 1:

```
Input: [2, 2, 1]
Output: 1
```

Example 2:

```
Input: [4, 1, 2, 1, 2]
Output: 4
```

Solution: XOR. This one is kinda straightforward. You'll need to know the properties of XOR as shown in Section 8.1.

```
1 n ^ n = 0
2 n ^ 0 = n
```

Therefore, we only need one variable to record the state which is initialize with 0: the first time to appear $x = n$, second time to appear $x = 0$. the last element x will be the single number. To set the statem we can use XOR.

```
1 def singleNumber(self, nums):
2     """
3     :type nums: List[int]
4     :rtype: int
5     """
6     v = 0
7     for e in nums:
8         v = v ^ e
9     return v
```

8.2 **137. Single Number II** Given a non-empty array of integers, every element appears three times except for one, which appears exactly once. Find that single one. *Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?*

1 Example 1:

```
2
3 Input: [2, 2, 3, 2]
4 Output: 3
```

5 Example 2:

```
7
8 Input: [0, 1, 0, 1, 0, 1, 99]
9 Output: 99
```

Solution: XOR and Two Variables. In this problem, because all element but one appears three times. To record the states of three, we need at least two variables. And we initialize it to $a = 0, b = 0$. For example, when 2 appears the first time, we set $a = 2, b = 0$; when it appears two times, $a = 0, b = 2$; when it appears three times, $a = 0, b = 0$. For number that appears one or two times will be saves either in a or in b . Same as the above example, we need to use XOR to change the state for each variable. We first do $a = a \text{ XOR } v, b = b \text{ XOR } v$, we need to keep a unchanged and set b to zero. We can do this as $a = a \text{ XOR } v \& \sim b; b = b \text{ XOR } v \& \sim a$.

```

1 def singleNumber(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: int
5     """
6     a = b = 0
7     for num in nums:
8         a = a ^ num & ~b
9         b = b ^ num & ~a
10    return a | b

```

8.3 421. Maximum XOR of Two Numbers in an Array (medium).

Given a non-empty array of numbers, $a_0, a_1, a_2, \dots, a_{n-1}$, where $0 \leq a_i < 2^{31}$. Find the maximum result of $a_i \text{ XOR } a_j$, where $0 \leq i, j < n$. Could you do this in $O(n)$ runtime?

Example :

Input: [3, 10, 5, 25, 2, 8]

Output: 28

Explanation: The maximum result is $5 \text{ } \wedge \text{ } 25 = 28$.

Solution 1: Build the Max bit by bit. First, let's convert these integers into binary representation by hand.

3	0000, 0011
10	0000, 1011
5	0000, 0101
25	0001, 1001
2	0000, 0010
8	0000, 1000

If we only look at the highest position i where there is one one and all others zero. Then we know the maximum XOR m has 1 at that bit. Now, we look at two bits: $i, i-1$. The possible maximum XOR for this is append 0 or 1 at the end of m , we have possible max 11, because for XOR, if we do XOR of m with others, $m \text{ XOR } a = b$, if b exists in these possible two sets, then max is possible and it become $m \ll 1 + 1$. We can carry on this process, the following process is showed as follows: answer $\hat{1}$ is the possible max,

```

1 def findMaximumXOR(self, nums):
2     """
3     :type nums: List[int]
4     :rtype: int
5     """
6     answer = 0
7     for i in range(32)[::-1]:
8         answer <= 1 # multiple it by two
9         prefixes = {num >> i for num in nums} # shift right
10        for n, divide/2^i, get the first (32-i) bits
11            answer += any((answer+1) ^ p in prefixes for p in
12                prefixes)
13        return answer

```

Solution 2: Use Trie.

```

1 def findMaximumXOR(self, nums):
2     def Trie():
3         return collections.defaultdict(Trie)
4
5     root = Trie()
6     best = 0
7
8     for num in nums:
9         candidate = 0
10        cur = this = root
11        for i in range(32)[::-1]:
12            curBit = num >> i & 1
13            this = this[curBit]
14            if curBit ^ 1 in cur:
15                candidate += 1 << i
16                cur = cur[curBit ^ 1]
17            else:
18                cur = cur[curBit]
19            best = max(candidate, best)
20    return best

```

With Mask

8.4 190. Reverse Bits (Easy). Reverse bits of a given 32 bits unsigned integer.

Example 1:

Input: 0000001010010100000111010011100
Output: 00111001011110000010100101000000
Explanation: The input binary string

0000001010010100000111010011100 represents the unsigned integer 43261596, so return 964176192 which its binary representation is 00111001011110000010100101000000.

Example 2:

```

Input: 1111111111111111111111111111111101
Output: 1011111111111111111111111111111111
Explanation: The input binary string
1111111111111111111111111111111101 represents the unsigned
integer 4294967293, so return 3221225471 which its
binary representation is
1010111110010110010011101101001.
```

Solution: Get Bit and Set bit with mask. We first get bits from the most significant position to the least significant position. And get the bit at that position with mask, and set the bit in our 'ans' with a mask indicates the position of (31-i):

```

1 # @param n, an integer
2 # @return an integer
3 def reverseBits(self, n):
4     ans = 0
5     for i in range(32)[::-1]: #from high to low
6         mask = 1 << i
7         set_mask = 1 << (31-i)
8         if (mask & n) != 0: #get bit
9             #set bit
10            ans |= set_mask
11    return ans
```

8.5 201. Bitwise AND of Numbers Range (medium). Given a range $[m, n]$ where $0 \leq m \leq n \leq 2147483647$, return the bitwise AND of all numbers in this range, inclusive.

Example 1:

```

Input: [5 ,7]
Output: 4
```

Example 2:

```

Input: [0 ,1]
Output: 0
```

Solution 1: O(n) do AND operation. We start a 32 bit long 1s. The solution would receive LTE error.

```

1 def rangeBitwiseAnd(self, m, n):
2     """
3     :type m: int
4     :type n: int
5     :rtype: int
6     """
7     ans = int('1'*32, 2)
8     for c in range(m, n+1):
9         ans &= c
10    return ans
```

Solution 2: Use mask, check bit by bit. Think, if we AND all, the resulting integer would definitely smaller or equal to m . For example 1:

```
0101 5
0110 6
0111 7
```

We start from the least significant bit at 5, if it is 1, then we check the closest number to 5 that has 0 at the this bit. It would be 0110. If this number is in the range, then this bit is offset to 0. We then move on to check the second bit. To make this closest number: first we clear the least $i+1$ positions in m to get 0100 and then we add it with $1 \ll (i + 1)$ as 0010 to get 0110.

```
1 def rangeBitwiseAnd(self, m, n):
2     ans = 0
3     mask = 1
4     for i in range(32): # [: - 1]:
5         bit = mask & m != 0
6         if bit:
7             # clear i+1, ..., 0
8             mask_clear = (mask << 1) - 1
9             left = m & (~mask_clear)
10            check_num = (mask << 1) + left
11            if check_num < m or check_num > n:
12                ans |= 1 << i
13            mask = mask << 1
14    return ans
```

Solution 3: Use While Loop. We can start do AND of n with $(n-1)$. If the resulting integer is still larger than m , then we keep do such AND operation.

```
1 def rangeBitwiseAnd(self, m, n):
2     ans=n
3     while ans>m:
4         ans=ans&(ans-1)
5     return ans
```

8.6 Exercises

1. Write a function to determine the number of bits required to convert integer A to integer B.

```
1 def bitswaprequired(a, b):
2     count = 0
3     c = a ^ b
4     while(c != 0):
5         count += c & 1
6         c = c >> 1
```

```

7     return count
8 print(bitswaprequired(12, 7))

```

2. **389. Find the Difference (easy).** Given two strings s and t which consist of only lowercase letters. String t is generated by random shuffling string s and then add one more letter at a random position. Find the letter that was added in t .

Example :

Input :
 $s = "abcd"$
 $t = "abcde"$

Output :

e

Explanation :

'e' is the letter that was added.

Solution 1: Use Counter Difference. This way we need $O(M+N)$ space to save the result of counter for each letter.

```

1 def findTheDifference(self, s, t):
2     s = collections.Counter(s)
3     t = collections.Counter(t)
4     diff = t - s
5     return list(diff.keys())[0]

```

Solution 2: Single Number with XOR. Using bit manipulation and with $O(1)$ we can find it in $O(M + N)$ time, which is the best BCR:

```

1 def findTheDifference(self, s, t):
2     """
3     :type s: str
4     :type t: str
5     :rtype: str
6     """
7     v = 0
8     for c in s:
9         v = v ^ ord(c)
10    for c in t:
11        v = v ^ ord(c)
12    return chr(v)

```

3. **50. Pow(x, n) (medium).** for n , such as 10, we represent it as 1010, if we have a base and an result, we start from the least significant position, each time we move, the base because $\text{base} * \text{base}$, and if the value if 1, then we multiple the answer with the base.

9

Python Data Structures

9.1 Introduction

Python is object-oriented programming language where each object is implemented using C++ in the backend. The built-in data types of C++ follows more rigidly to the abstract data structures. We would get by just learning how to use Python data types alone: its **property**—immutable or mutable, its **built-in in-place operations**—such as `append()`, `insert()`, `add()`, `remove()`, `replace()` and so, and **built-in functions** and **operations** that offers additional ability to manipulate data structure—an object here. However, some data types' behaviors might confuse us with abstract data structures, making it hard to access and evaluate its efficiency.

In this chapter and the following three chapters, we starts to learn Python data structures by relating its C++ data structures to our learned abstract data structures, and then introduce each's property, built-in operations, built-in functions and operations. *Please read the section *Understanding Object in the Appendix–Python Knowledge Base* to to study the properties of Built-in Data Types first if Python is not your familiar language.*

Python Built-in Data Types In Python 3, we have four built-in scalar data types: `int`, `float`, `complex`, `bool`. At higher level, it includes four sequence types: `str`—string type, `list`, `tuple`, and `range`; one mapping type: `dict` and two set types: `set` and `frozenset`. Among these 12 built-in data types, other than the scalar types, the others representing some of our introduced abstract data structures.

Abstract Data Types with Python Data Types/Modules To relate the abstract data types to our build-in data types we have:

- Sequence type corresponds to Array data structure: includes `string`, `list`, `tuple`, and `range`
- `dict`, `set`, and `frozensest` maps to the hash tables.
- For linked list, stack, queue, we either need to implement it with build-in data types or we have Python Modules.

9.2 Array and Python Sequence

We will see from other remaining contents of this part that how array-based Python data structures are used to implement the other data structures. On the LeetCode, these two data structures are involved into 25% of LeetCode Problems.

9.2.1 Introduction to Python Sequence

In Python, *sequences* are defined as ordered sets of objects indexed by non-negative integers, we use *index* to refer and in Python it defaultly starts at 0. Sequence types are **iterable**. Iterables are able to be iterated over. Iterators are the agents that perform the iteration, where we have `iter()` built-in function.

- `string` is a sequence of characters, it is immutable, and with **static array** as its backing data structure in C++.
- `list` and `tuple` are sequences of **arbitrary** objects.–meaning it accepts different types of objects including the 12 built-in data types and any other objects. This sounds fancy and like magic! However, it does not change the fact that its backing abstract data structure is **dynamic array**. They are able to have arbitrary type of objects through the usage of pointers to objects, pointing to object's physical location, and each pointer takes fixed number of bytes in space (in 32-bit system, 4 bytes, and for a 64-bit system, 8 bytes instead).
- `range`: In Python 3, `range()` is a type. But range does not have backing array data structure to save a sequence of value, it computes on demand. Thus we will first introduce range and get done with it before we focus on other sequence types.

```

1 >>> type(range)
2 <class 'type'>

```

All these sequence type data structures share the most common methods and operations shown in Table 9.4 and 9.5. To note that in Python, the indexing starts from 0.

Let us examine each type of sequence further to understand its performance, and relation to array data structures.

9.2.2 Range

Range Syntax

The range object has three attributes: `start`, `stop`, `step`, and a range object can be created as `range(start, stop, step)`. These attributes need to integers—both negative and positive works—to define a range, which will be $[start, stop)$. The default value for `start` and `stop` is 0. For example:

```

1 >>> a = range(10)
2 >>> b = range(0, 10, 2)
3 >>> a, b
4 (range(0, 10), range(0, 10, 2))
```

Now, we print it out:

```

1 >>> for i in a:
2 ...     print(i, end=' ')
3 ...
4 0 1 2 3 4 5 6 7 8 9
```

And for `b`, it will be:

```

1 >>> for i in b:
2 ...     print(i, end=' ')
3 ...
4 0 2 4 6 8
```

Like any other sequence types, `range` is iterable, can be indexed and sliced.

What you do not see

The range object might be a little bizarre when we first learn it. Is it an iterator, a generator? The answer to both questions are NO. What is it then? It is more like a sequence type that differs itself without other counterparts with its own unique properties:

- It is “lazy” in the sense that it doesn’t generate every number that it “contain” when we create it. Instead it gives those numbers to us as we need them when looping over it. Thus, it saves us space:

```

1 >>> a = range(1_000_000)
2 >>> b = [i for i in a]
3 >>> a.__sizeof__(), b.__sizeof__()
4 (48, 8697440)
```

This is just how we define the behavior of the range class back in the C++ code. We does not need to save all integers in the range, but be generated with function that specifically asks for it.

- It is not an iterator; it won't get consumed. We can iterate it multiple times. This is understandable given how it is implemented.

9.2.3 String

String is **static array** and its items are just characters, represented using ASCII or Unicode¹. String is immutable which means once its created we can no longer modify its content or extent its size. String is more compact compared with storing the characters in **list** because of its backing array wont be assigned to any extra space.

String Syntax

strings can be created in Python by wrapping a sequence of characters in single or double quotes. Multi-line strings can easily be created using three quote characters.

New a String We specially introduce some commonly and useful functions.

Join The **str.join()** method will concatenate two strings, but in a way that passes one string through another. For example, we can use the **str.join()** method to add whitespace to that string, which we can do like so:

```
1 balloon = "Sammy has a balloon."
2 print(" ".join(balloon))
3 #Output
4 S a m m y   h a s     a     b a l l o o n .
```

The **str.join()** method is also useful to combine a list of strings into a new single string.

```
1 print(" ".join(["a", "b", "c"]))
2 #Output
3 abc
```

¹In Python 3, all strings are represented in Unicode. In Python 2 are stored internally as 8-bit ASCII, hence it is required to attach 'u' to make it Unicode. It is no longer necessary now.

Split Just as we can join strings together, we can also split strings up using the `str.split()` method. This method separates the string by whitespace if no other parameter is given.

```
1 print(balloon.split())
2 #Ouput
3 ['Sammy', 'has', 'a', 'balloon.']}
```

We can also use `str.split()` to remove certain parts of an original string. For example, let's remove the letter 'a' from the string:

```
1 print(balloon.split("a"))
2 #Ouput
3 ['S', 'mmy h', 's ', ' b', 'lloon. ']
```

Now the letter a has been removed and the strings have been separated where each instance of the letter a had been, with whitespace retained.

Replace The `str.replace()` method can take an original string and return an updated string with some replacement.

Let's say that the balloon that Sammy had is lost. Since Sammy no longer has this balloon, we will change the substring "has" from the original string balloon to "had" in a new string:

```
1 print(balloon.replace("has", "had"))
2 #Ouput
3 Sammy had a balloon.
```

We can use the replace method to delete a substring:

```
1 balloon.replace("has", "")
```

Using the string methods `str.join()`, `str.split()`, and `str.replace()` will provide you with greater control to manipulate strings in Python.

Conversion between Integer and Character Function `ord()` would get the int value (ASCII) of the char. And in case you want to convert back after playing with the number, function `chr()` does the trick.

```
1 print(ord('A'))# Given a string of length one, return an integer
                 representing the Unicode code point of the character when
                 the argument is a unicode object,
2 print(chr(65))
```

String Functions

Because string is one of the most fundamental built-in data types, this makes managing its built-in common methods shown in Table 9.1 and 9.2 necessary. Use boolean methods to check whether characters are lower case, upper case, or title case, can help us to sort our data appropriately, as well as provide us with the opportunity to standardize data we collect by checking and then modifying strings as needed.

Table 9.1: Common Methods of String

Method	Description
count(substr, [start, end])	Counts the occurrences of a substring with optional start and end position
find(substr, [start, end])	Returns the index of the first occurrence of a substring or returns -1 if the substring is not found
join(t)	Joins the strings in sequence t with current string between each item
lower()/upper()	Converts the string to all lowercase or uppercase
replace(old, new)	Replaces old substring with new substring
strip([characters])	Removes whitespace or optional characters
split([characters], [maxsplit])	Splits a string separated by whitespace or an optional separator. Returns a list
expandtabs([tabsize])	Replaces tabs with spaces.

Table 9.2: Common Boolean Methods of String

Boolean Method	Description
isalnum()	String consists of only alphanumeric characters (no symbols)
isalpha()	String consists of only alphabetic characters (no symbols)
islower()	String's alphabetic characters are all lower case
isnumeric()	String consists of only numeric characters
isspace()	String consists of only whitespace characters
istitle()	String is in title case
isupper()	String's alphabetic characters are all upper case

9.2.4 List

The underlying abstract data structure of `list` data types is **dynamic array**, meaning we can add, delete, modify items in the list. It supports random access by indexing. List is the most widely one among sequence types due to its mutability.

Even if list supports data of arbitrary types, we do not prefer to do this. Use `tuple` or `namedtuple` for better practice and offers better clarification.

What You see: List Syntax

New a List: We have multiple ways to new either empty list or with initialized data. List comprehension is an elegant and concise way to create new list from an existing list in Python.

```
1 # new an empty list
```

```

2 lst = []
3 lst2 = [2, 2, 2, 2] # new a list with initialization
4 lst3 = [3]*5      # new a list size 5 with 3 as initialization
5 print(lst, lst2, lst3)
6 # output
7 # [] [2, 2, 2, 2] [3, 3, 3, 3]

```

We can use **list comprehension** and use `enumerate` function to loop over its items.

```

1 lst1 = [3]*5      # new a list size 5 with 3 as initialization
2 lst2 = [4 for i in range(5)]
3 for idx, v in enumerate(lst1):
4     lst1[idx] += 1

```

Search We use method `list.index()` to obtain the index of the searched item.

```

1 print(lst.index(4)) #find 4, and return the index
2 # output
3 # 3

```

If we `print(lst.index(5))` will raise `ValueError: 5 is not in list`. Use the following code instead.

```

1 if 5 in lst:
2     print(lst.index(5))

```

Add Item We can add items into list through `insert(index, value)`—inserting an item at a position in the original list or `list.append(value)`—appending an item at the end of the list.

```

1 # INSERTION
2 lst.insert(0, 1) # insert an element at index 0, and since it is
                  # empty lst.insert(1, 1) has the same effect
3 print(lst)
4
5 lst2.insert(2, 3)
6 print(lst2)
7 # output
8 # [1]
9 # [2, 2, 3, 2, 2]
10 # APPEND
11 for i in range(2, 5):
12     lst.append(i)
13 print(lst)
14 # output
15 # [1, 2, 3, 4]

```

Delete Item

Get Size of the List We can use `len` built-in function to find out the number of items storing in the list.

```

1 print(len(lst2))
2 # 4

```

What you do not see: Understand List

To understand list, we need start with its C++ implementation, we do not introduce the C++ source code, but instead use function to access and evaluate its property.

List Object and Pointers In a 64-bits (8 bytes) system, such as in Google Colab, a pointer is represented with 8 bytes space. In Python3, the list object itself takes 64 bytes in space. And any additional element takes 8 bytes. In Python, we can use `getsizeof()` from `sys` module to get its memory size, for example:

```
1 lst_lst = [[], [1], ['1'], [1, 2], ['1', '2']]
```

And now, let us get the memory size of `lst_lst` and each list item in this list.

```
1 import sys
2 for lst in lst_lst:
3     print(sys.getsizeof(lst), end=' ')
4 print(sys.getsizeof(lst_lst))
```

The output is:

```
64 72 72 80 80 104
```

We can see a list of integers takes the same memory size as of a list of strings with equal length.

insert and append Whenever `insert` and `append` is called, and assume the original length is n , Python could compare $n + 1$ with its allocated length. If you append or insert to a Python list and the backing array isn't big enough, the backing array must be expanded. When this happens, the backing array is grown by approximately 12% the following formula (comes from C++):

```
1 new_allocated = (size_t)newsize + (newsize >> 3) +
2     (newsize < 9 ? 3 : 6);
```

Do an experiment, we can see how it works. Here we use `id()` function to obtain the pointer's physical address. We compare the size of the list and its underlying backing array's real additional size in space (with 8 bytes as unit).

```
1 a = []
2 for size in range(17):
3     a.insert(0, size)
4     print('size:', len(a), 'bytes:', (sys.getsizeof(a)-64)//8, 'id'
      :, id(a))
```

The output is:

```

size: 1 bytes: 4 id: 140682152394952
size: 2 bytes: 4 id: 140682152394952
size: 3 bytes: 4 id: 140682152394952
size: 4 bytes: 4 id: 140682152394952
size: 5 bytes: 8 id: 140682152394952
size: 6 bytes: 8 id: 140682152394952
size: 7 bytes: 8 id: 140682152394952
size: 8 bytes: 8 id: 140682152394952
size: 9 bytes: 16 id: 140682152394952
size: 10 bytes: 16 id: 140682152394952
size: 11 bytes: 16 id: 140682152394952
size: 12 bytes: 16 id: 140682152394952
size: 13 bytes: 16 id: 140682152394952
size: 14 bytes: 16 id: 140682152394952
size: 15 bytes: 16 id: 140682152394952
size: 16 bytes: 16 id: 140682152394952
size: 17 bytes: 25 id: 140682152394952

```

The output addresses the growth patterns as [0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...].

Amortizely, `append` takes $O(1)$. However, it is $O(n)$ for `insert` because it has to first shift all items in the original list from $[pos, end]$ by one position, and put the item at pos with random access.

Common Methods of List

We have already seen how to use `append`, `insert`. Now, Table 9.3 shows us the common List Methods, and they will be used as `list.methodName()`.

Table 9.3: Common Methods of List

Method	Description
<code>append()</code>	Add an element to the end of the list
<code>extend(l)</code>	Add all elements of a list to the another list
<code>insert(index, val)</code>	Insert an item at the defined index s
<code>pop(index)</code>	Removes and returns an element at the given index
<code>remove(val)</code>	Removes an item from the list
<code>clear()</code>	Removes all items from the list
<code>index(val)</code>	Returns the index of the first matched item
<code>count(val)</code>	Returns the count of number of items passed as an argument
<code>sort()</code>	Sort items in a list in ascending order
<code>reverse()</code>	Reverse the order of items in the list (same as <code>list[::-1]</code>)
<code>copy()</code>	Returns a shallow copy of the list (same as <code>list[:]</code>)

Two-dimensional List

Two dimensional list is a list within a list. In this type of array the position of an data element is referred by two indices instead of one. So it represents a table with rows and columns of data. For example, we can declare the following 2-d array:

```
1 ta = [[11, 3, 9, 1], [25, 6, 10], [10, 8, 12, 5]]
```

The scalar data in two dimensional lists can be accessed using two indices. One index referring to the main or parent array and another index referring to the position of the data in the inner list. If we mention only one index then the entire inner list is printed for that index position. The example below illustrates how it works.

```
1 print(ta[0])
2 print(ta[2][1])
```

And with the output

```
[11, 3, 9, 1]
8
```

In the above example, we new a 2-d list and initialize them with values. There are also ways to new an empty 2-d array or fix the dimension of the outer array and leave it empty for the inner arrays:

```
1 # empty two dimensional list
2 empty_2d = []
3
4 # fix the outer dimension
5 fix_out_d = [[] for _ in range(5)]
6 print(fix_out_d)
```

All the other operations such as delete, insert, update are the same as of the one-dimensional list.

Matrices We are going to need the concept of matrix, which is defined as a collection of numbers arranged into a fixed number of rows and columns. For example, we define 3×4 (read as 3 by 4) order matrix is a set of numbers arranged in 3 rows and 4 columns. And for m_1 and m_2 , they are doing the same things.

```
1 rows, cols = 3, 4
2 m1 = [[0 for _ in range(cols)] for _ in range(rows)] # rows *
3           cols
4 m2 = [[0]*cols for _ in range(rows)] # rows * cols
5 print(m1, m2)
```

The output is:

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]] [[0, 0, 0, 0], [0, 0,
0, 0], [0, 0, 0, 0]]
```

We assign value to m1 and m2 at index (1, 2) with value 1:

```

1 m1[1][2] = 1
2 m2[1][2] = 1
3 print(m1, m2)

```

And the output is:

```
[ [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0] ] [[0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]]
```

However, we can not declare it in the following way, because we end up with some copies of the same inner lists, thus modifying one element in the inner lists will end up changing all of them in the corresponding positions. Unless the feature suits the situation.

```

1 # wrong declaration
2 m4 = [[0]*cols]*rows
3 m4[1][2] = 1
4 print(m4)

```

With output:

```
[ [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0] ]
```

Access Rows and Columns In the real problem solving, we might need to access rows and columns. Accessing rows is quite easy since it follows the declaration of two-dimensional array.

```

1 # accessing row
2 for row in m1:
3     print(row)

```

With the output:

```
[0, 0, 0, 0]
[0, 0, 1, 0]
[0, 0, 0, 0]
```

However, accessing columns will be less straightforward. To get each column, we need another inner for loop or list comprehension through all rows and obtain the value from that column. This is usually a lot slower than accessing each row due to the fact that each row is a pointer while each col we need to obtain from each row.

```

1 # accessing col
2 for i in range(cols):
3     col = [row[i] for row in m1]
4     print(col)

```

The output is:

```
[0, 0, 0]
[0, 0, 0]
[0, 1, 0]
[0, 0, 0]
```

There's also a handy "idiom" for transposing a nested list, turning 'columns' into 'rows':

```
1 transposedM1 = list(zip(*m1))
2 print(transposedM1)
```

The output will be:

```
[(0, 0, 0), (0, 0, 0), (0, 1, 0), (0, 0, 0)]
```

9.2.5 Tuple

A **tuple** has **static array** as its backing abstract data structure in C, which is immutable—we can not add, delete, or replace items once its created and assigned with value. You might think if **list** is a dynamic array and has no restriction same as of the tuple, why would we need **tuple** then?

Tuple VS List We list how we use each data type and why is it. The main benefit of tuple's immutability is it is hashable, we can use them as keys in the hash table—**dictionary types**, whereas the mutable types such as list and range can not be applied. Besides, in the case that the data does not change, the tuple's immutability will guarantee that the data remains write-protected and iterating an immutable sequence is faster than a mutable sequence, giving it slight performance boost. Also, we generally use tuple to store a variety of data types. For example, in a class score system, for a student, we might want to have its name, student id, and test score, we can write ('Bob', 12345, 89).

Tuple Syntax

New and Initialize Tuple Tuples are created by separating the items with a comma. It is commonly wrapped in parentheses for better readability. Tuple can also be created via a built-in function **tuple()**, if the argument to **tuple()** is a sequence then this creates a tuple of elements of that sequences. This is also used to realize type conversion.

An empty tuple:

```
1 tup = ()
2 tup3 = tuple()
```

When there is only one item, put comma behind so that it won't be translated as **string**, which is a bit bizarre!

```
1 tup2 = ('crack', )
2 tup1 = ('crack', 'leetcode', 2018, 2019)
```

Converting a string to a tuple with each character separated.

```
1 tup4 = tuple("leetcode") # the sequence is passed as a tuple of
                           # elements
2 >> tup4: ('l', 'e', 'e', 't', 'c', 'o', 'd', 'e')
```

Converting a list to a tuple.

```
1 tup5 = tuple(['crack', 'leetcode', 2018, 2019]) # same as tuple1
```

If we print out these tuples, it will be

```
1 tup1: ('crack', 'leetcode', 2018, 2019)
2 tup2: crack
3 tup3: ()
4 tup4: ('l', 'e', 'e', 't', 'c', 'o', 'd', 'e')
5 tup5: ('crack', 'leetcode', 2018, 2019)
```

Changing a Tuple Assume we have the following tuple:

```
1 tup = ('a', 'b', [1, 2, 3])
```

If we want to change it to ('c', 'b', [4,2,3]). We can not do the following operation as we said a tuple cannot be changed in-place once it has been assigned.

```
1 tup = ('a', 'b', [1, 2, 3])
2 #tup[0] = 'c' #TypeError: 'tuple' object does not support item assignment
```

Instead, we initialize another tuple and assign it to `tup` variable.

```
1 tup=( 'c', 'b', [4,2,3])
```

However, for its items which are mutable itself, we can still manipulate it. For example, we can use index to access the list item at the last position of a tuple and modify the list.

```
1 tup[-1][0] = 4
2 #('a', 'b', [4, 2, 3])
```

Understand Tuple

The backing structure is **static array** which states that the way the tuple is structure is similar to list, other than its write-protected. We will just brief on its property.

Tuple Object and Pointers Tuple object itself takes 48 bytes. And all the others are similar to corresponding section in list.

```
1 lst_tup = [(), (1,), ('1',), (1, 2), ('1', '2')]
2 import sys
3 for tup in lst_tup:
4     print(sys.getsizeof(tup), end=' ')
```

The output will be:

```
48 56 56 64 64
```

Named Tuples

In named tuple, we can give all records a name, say “Computer_Science” to indicate the class name, and we give each item a name, say ‘name’, ‘id’, and ‘score’. We need to import `namedtuple` class from module `collections`. For example:

```

1 record1 = ('Bob', 12345, 89)
2 from collections import namedtuple
3 Record = namedtuple('Computer_Science', 'name id score')
4 record2 = Record('Bob', id=12345, score=89)
5 print(record1, record2)
```

The output will be:

```
1 ('Bob', 12345, 89) Computer_Science(name='Bob', id=12345, score
=89)
```

9.2.6 Summary

All these sequence type data structures share the most common methods and operations shown in Table 9.4 and 9.5. To note that in Python, the indexing starts from 0.

9.2.7 Bonus

Circular Array The corresponding problems include:

1. 503. Next Greater Element II

9.2.8 Exercises

1. 985. Sum of Even Numbers After Queries (easy)
2. 937. Reorder Log Files

You have an array of logs. Each log is a space delimited string of words.

For each log, the first word in each log is an alphanumeric identifier. Then, either:

Each word after the identifier will consist only of lowercase letters, or;
Each word after the identifier will consist only of digits.

We will call these two varieties of logs letter-logs and digit-logs. It is guaranteed that each log has at least one word after its identifier.

Reorder the logs so that all of the letter-logs come before any digit-log. The letter-logs are ordered lexicographically ignoring identifier, with the identifier used in case of ties. The digit-logs should be put in their original order.

Return the final order of the logs.

Table 9.4: Common Methods for Sequence Data Type in Python

Function Method	Description
len(s)	Get the size of sequence s
min(s, [default=obj, key=func])	The minimum value in s (alphabetically for strings)
max(s, [default=obj, key=func])	The maximum value in s (alphabetically for strings)
sum(s, [start=0])	The sum of elements in s (return <i>TypeError</i> if s is not numeric)
all(s)	Return <i>True</i> if all elements in s are True (Similar to <i>and</i>)
any(s)	Return <i>True</i> if any element in s is True (similar to <i>or</i>)

Table 9.5: Common out of place operators for Sequence Data Type in Python

Operation	Description
s + r	Concatenates two sequences of the same type
s * n	Make n copies of s, where n is an integer
v ₁ , v ₂ , ..., v _n = s	Unpack n variables from s
s[i]	Indexing-returns i th element of s
s[i:j:stride]	Slicing-returns elements between i and j with optimal stride
x in s	Return <i>True</i> if element x is in s
x not in s	Return <i>True</i> if element x is not in s

```

1 Example 1:
2
3 Input: ["a1 9 2 3 1","g1 act car","zo4 4 7","ab1 off key
      dog","a8 act zoo"]
4 Output: ["g1 act car","a8 act zoo","ab1 off key dog","a1 9
      2 3 1","zo4 4 7"]
5
6
7
8 Note:
9
10    0 <= logs.length <= 100
11    3 <= logs[i].length <= 100
12    logs[i] is guaranteed to have an identifier , and a word
        after the identifier .

```

```

1 def reorderLogFiles(self, logs):
2     letters = []
3     digits = []
4     for idx, log in enumerate(logs):
5         splited = log.split(' ')
6         id = splited[0]
7         type = splited[1]

```

```
9     if type.isnumeric():
10        digits.append(log)
11    else:
12        letters.append(( ' '.join(splited[1:]) , id))
13 letters.sort() #default sorting by the first element
14 and then the second in the tuple
15
16 return [id + ' ' + other for other , id in letters] +
17 digits
```



```
1 def reorderLogFile(logs):
2     digit = []
3     letters = []
4     info = {}
5     for log in logs:
6         if '0' <= log[-1] <= '9':
7             digit.append(log)
8         else:
9             letters.append(log)
10            index = log.index(' ')
11            info[log] = log[index+1:]
12
13 letters.sort(key= lambda x: info[x])
14 return letters + digit
```

9.3 Linked List

Python does not have built-in data type or modules that offers the Linked List-like data structures, however, it is not hard to implement it ourselves.

9.3.1 Singly Linked List

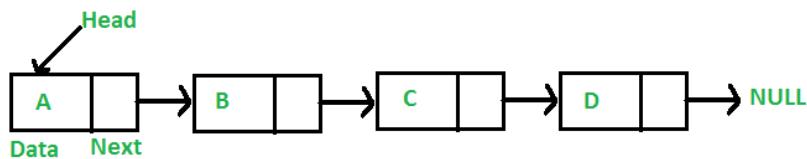


Figure 9.1: Linked List Structure

Linked list consists of **nodes**, and each **node** consists of at least two variables for singly linked list: **val** to save data and **next**, a pointer that points to the successive node. The **Node** class is given as:

```
1 class Node( object ):  
2     def __init__( self , val = None ):  
3         self.val = val
```

```
4     self.next = None
```

In Singly Linked List, usually we can start to with a **head** node which points to the first node in the list; only with this single node we are able to trace other nodes. For simplicity, demonstrate the process without using class, but we provide a class implementation with name **SinglyLinkedList** in our online python source code. Now, let us create an empty node named **head**.

```
1 head = None
```

We need to implement its standard operations, including insertion/append, delete, search, clear. However, if we allow to the head node to be **None**, there would be special cases to handle. Thus, we implement a **dummy node**—a node but with **None** as its value as the head, to simplify the coding. Thus, we point the head to a dummy node:

```
1 head = Node(None)
```

Append Operation As the append function in list, we add node at the very end of the linked list. If without the dummy node, then there will be two cases:

- When **head** is an empty node, we assign the new **node** to **head**.
- When it is not empty, we because all we have that is available is the head pointer, thus, it we need to first traverse all the nodes up till the very last node whose **next** is **None**, then we connect **node** to the last node through assigning it to the last node's **next** pointer.

The first case is simply bad: we would generate a new node and we can not track the head through in-place operation. However, with the dummy node, only the second case will appear. The code is:

```
1 def append(head, val):
2     node = Node(val)
3     cur = head
4     while cur.next:
5         cur = cur.next
6     cur.next = node
7     return
```

Now, let use create the same exact linked list in Fig. 9.1:

```
1 for val in [ 'A' , 'B' , 'C' , 'D' ]:
2     append(head, val)
```

Generator and Search Operations In order to traverse and iterate the linked list using syntax like `for ... in` statement like any other sequence data types in Python, we implement the `gen()` function that returns a generator of all nodes of the list. Because we have a dummy node, so we always start at `head.next`.

```
1 def gen(head):
2     cur = head.next
3     while cur:
4         yield cur
5         cur = cur.next
```

Now, let us print out the linked list we created:

```
1 for node in iter(head):
2     print(node.val, end=' ')
```

Here is the output:

```
A B C D
```

Search operation we find a node by value, and we return this node, otherwise, we return `None`.

```
1 def search(head, val):
2     for node in gen(head):
3         if node.val == val:
4             return node
5     return None
```

Now, we search for value 'B' with:

```
1 node = search(head, 'B')
```

Delete Operation For deletion, there are two scenarios: deleting a node by value when we are given the head node and deleting a given node such as the node we got from searching 'B'.

The first case requires us to first locate the node first, and rewire the pointers between the predecessor and successor of the deleting node. Again here, if we do not have a dummy node, we would have two cases: if the node is the head node, repoint the head to the next node, we connect the previous node to deleting node's next node, and the head pointer remains untouched. With dummy node, we would only have the second situation. In the process, we use an additional variable `prev` to track the predecessor.

```
1 def delete(head, val):
2     cur = head.next # start from dummy node
3     prev = head
4     while cur:
5         if cur.val == val:
6             # rewire
7             prev.next = cur.next
8             return
```

```

9     prev = cur
10    cur = cur.next

```

Now, let us delete one more node—'A' with this function.

```

1 delete(head, 'A')
2 for n in gen(head):
3     print(n.val, end = ' ')

```

Now the output will indicate we only have two nodes left:

```

1 C D

```

The second case might seems a bit impossible—we do not know its previous node, the trick we do is to copy the value of the next node to current node, and we delete the next node instead by pointing current node to the node after next node. While, that is only when the deleting node is not the last node. When it is, we have no way to completely delete it; but we can make it “invalid” by setting value and `Next` to `None`.

```

1 def delete(head, val):
2     cur = head.next # start from dummy node
3     prev = head
4     while cur:
5         if cur.val == val:
6             # rewire
7             prev.next = cur.next
8             return
9         prev = cur
10        cur = cur.next

```

Now, let us try deleting the node 'B' via our previously found `node`.

```

1 deleteByNode(node)
2 for n in gen(head):
3     print(n.val, end = ' ')

```

The output is:

```

1 A C D

```

Clear When we need to clear all the nodes of the linked list, we just set the node next to the dummy head to `None`.

```

1 def clear(self):
2     self.head = None
3     self.size = 0

```

Question: Some linked list can only allow insert node at the tail which is Append, some others might allow insertion at any location. To get the length of the linked list easily in $O(1)$, we need a variable to track the size

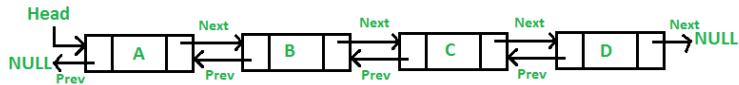


Figure 9.2: Doubly Linked List

9.3.2 Doubly Linked List

On the basis of Singly linked list, doubly linked list (dll) contains an extra pointer in the node structure which is typically called `prev` (short for previous) and points back to its predecessor in the list. We define the `Node` class as:

```

1 class Node:
2     def __init__(self, val, prev = None, next = None):
3         self.val = val
4         self.prev = prev # reference to previous node in DLL
5         self.next = next # reference to next node in DLL
  
```

Similarly, let us start with setting the dummy node as head:

```

1 head = Node()
  
```

Now, instead of for me to continue to implement all operations that are slightly variants of the singly linked list, why do not you guys implement it? Do not worry, try it first, and also I have the answer covered in the google colab, enjoy!

Now, I assume that you have implemented those operations and or checked up the solutions. We would notice in `search()` and `gen()`, the code is exactly the same, and for other operations, there is only one or two lines of code that differs from SLL. Let's quickly list these operations:

Append Operation In DLL, we have to set the appending node's `prev` pointer to the last node of the linked list. The code is:

```

1 def append(head, val):
2     node = Node(val)
3     cur = head
4     while cur.next:
5         cur = cur.next
6     cur.next = node
7     node.prev = cur ## only difference
8     return
  
```

Generator and Search Operations There is no much difference if we just search through `next` pointer. However, with the extra `prev` pointer, we can have two options: either search forward through `next` or backward

through `prev` if the given starting node is any node. Whereas for SLL, this is not an option, because we would not be able to conduct a complete search—we can only search among the items behind from the given node. When the data is ordered in some way, or if the program is parallel—situations that bidirectional search would make sense.

```

1 def gen(head):
2     cur = head.next
3     while cur:
4         yield cur
5         cur = cur.next

1 def search(head, val):
2     for node in gen(head):
3         if node.val == val:
4             return node
5     return None

```

Delete Operation To delete a node by value, we first find it in the linked list, and the rewiring process needs to deal with the next node's `prev` pointer if the next node exists.

```

1 def delete(head, val):
2     cur = head.next # start from dummy node
3     while cur:
4         if cur.val == val:
5             # rewire
6             cur.prev.next = cur.next
7             if cur.next:
8                 cur.next.prev = cur.prev
9             return
10            cur = cur.next

```

For `deleteByNode`, because we are cutting off `node.next`, we need to connect node to `node.next.next` in two directions: first point `prev` of later node to current node, and set point current node's `next` to the later node.

```

1 def deleteByNode(node):
2     # pull the next node to current node
3     if node.next:
4         node.val = node.next.val
5         if node.next.next:
6             node.next.next.prev = node
7             node.next = node.next.next
8         else: #last node
9             node.prev.next = None
10            return node

```

Comparison We can see there is some slight advantage of dll over sll, but it comes with the cost of handing the extra `prev`. This would only be an advantage when bidirectional searching plays dominant factor in the matter of efficiency, otherwise, better stick with sll.

Tips From our implementation, in some cases we still need to worry about if it is the last node or not. The coding logic can further be simplified if we put a dummy node at the end of the linked list too.

9.3.3 Bonus

Circular Linked List A circular linked list is a variation of linked list in which the first node connects to last node. To make a circular linked list from a normal linked list: in singly linked list, we simply set the last node's `next` pointer to the first node; in doubly linked list, other than setting the last node's `next` pointer, we set the `prev` pointer of the first node to the last node making the circular in both directions.

Compared with a normal linked list, circular linked list saves time for us to go to the first node from the last (both sll and dll) or go to the last node from the first node (in dll) by doing it in a single step through the extra connection. Because it is a circle, when ever a search with a `while` loop is needed, we need to make sure the end condition: just make sure we searched a whole cycle by comparing the iterating node to the starting node.

Recursion Recursion offers additional pass of traversal—bottom-up on the basis of the top-down direction and in practice, it offers clean and simpler code compared with iteration.

9.3.4 Hands-on Examples

Remove Duplicates (L83) Given a sorted linked list, delete all duplicates such that each element appear only once.

Example 1:

```
Input : 1->1->2
Output : 1->2
```

Example 2:

```
Input : 1->1->2->3->3
Output : 1->2->3
```

Analysis

This is a linear complexity problem, the most straightforward way is to iterate through the linked list and compare the current node's value with the next's to check its equivalency: (1) if YES: delete one of the nodes, here we go for the next node; (2) if NO: we can move to the next node safely and sound.

Iteration without Dummy Node We start from the `head` in a `while` loop, if the next node exists and if the value equals, we delete next node. However, after the deletion, we can not move to next directly; say if we have $1 \rightarrow 1 \rightarrow 1$, when the second 1 is removed, if we move, we will be at the last 1, and would fail removing all possible duplicates. The code is given:

```

1 def deleteDuplicates(self, head):
2     """
3         :type head: ListNode
4         :rtype: ListNode
5     """
6     if not head:
7         return None
8
9     def iterative(head):
10        current = head
11        while current:
12            if current.next and current.val == current.next.val:
13                # delete next
14                current.next = current.next.next
15            else:
16                current = current.next
17        return head
18
19    return iterative(head)

```

With Dummy Node We see with a dummy node, we put `current.next` in the whole loop, because only if the next node exists, would we need to compare the values. Besides, we do not need to check this condition within the `while` loop.

```

1 def iterative(head):
2     dummy = ListNode(None)
3     dummy.next = head
4     current = dummy
5     while current.next:
6         if current.val == current.next.val:
7             # delete next
8             current.next = current.next.next
9         else:
10             current = current.next
11     return head

```

Recursion Now, if we use recursion and return the node, thus, at each step, we can compare our node with the returned node (locating behind the current node), same logic applies. A better way to help us is drawing out an example. With $1 \rightarrow 1 \rightarrow 1$. The last 1 will return, and at the second last 1, we can compare them, because it equals, we delete the last 1, now we backtrack to the first 1 with the second last 1 as returned node, we compare again. The code is the simplest among all solutions.

```

1 def recursive(node):
2     if node.next is None:
3         return node
4
5     next = recursive(node.next)
6     if next.val == node.val:
7         node.next = node.next.next
8     return node

```

9.3.5 Exercises

Basic operations:

1. 237. Delete Node in a Linked List (easy, delete only given current node)
2. 2. Add Two Numbers (medium)
3. 92. Reverse Linked List II (medium, reverse in one pass)
4. 83. Remove Duplicates from Sorted List (easy)
5. 82. Remove Duplicates from Sorted List II (medium)
6. Sort List
7. Reorder List

Fast-slow pointers:

1. 876. Middle of the Linked List (easy)
2. Two Pointers in Linked List
3. Merge K Sorted Lists

Recursive and linked list:

1. 369. Plus One Linked List (medium)

9.4 Stack and Queue

Stack data structures fits well for tasks that require us to check the previous states from closest level to furthest level. Here are some exemplary applications: (1) reverse an array, (2) implement DFS iteratively as we will see in Chapter ??, (3) keep track of the return address during function calls, (4) recording the previous states for backtracking algorithms.

Queue data structures can be used: (1) implement BFS shown in Chapter ??, (2) implement queue buffer.

In the remaining section, we will discuss the implement with the built-in data types or using built-in modules. After this, we will learn more advanced queue and stack: the priority queue and the monotone queue which can be used to solve medium to hard problems on LeetCode.

9.4.1 Basic Implementation

For Queue and Stack data structures, the essential operations are two that adds and removes item. In Stack, they are usually called **PUSH** and **POP**. PUSH will add one item, and POP will remove one item and return its value. These two operations should only take $O(1)$ time. Sometimes, we need another operation called PEEK which just return the element that can be accessed in the queue or stack without removing it. While in Queue, they are named as **Enqueue** and **Dequeue**.

The simplest implementation is to use Python List by function *insert()* (insert an item at appointed position), *pop()* (removes the element at the given index, updates the list , and return the value. The default is to remove the last item), and *append()*. However, the list data structure can not meet the time complexity requirement as these operations can potentially take $O(n)$. We feel its necessary because the code is simple thus saves you from using the specific module or implementing a more complex one.

Stack The implementation for stack is simplily adding and deleting element from the end.

```

1 # stack
2 s = []
3 s.append(3)
4 s.append(4)
5 s.append(5)
6 s.pop()
```

Queue For queue, we can append at the last, and pop from the first index always. Or we can insert at the first index, and use pop the last element.

```

1 # queue
2 # 1: use append and pop
3 q = []
4 q.append(3)
5 q.append(4)
6 q.append(5)
7 q.pop(0)
```

Running the above code will give us the following output:

```

1 print('stack:', s, 'queue:', q)
2 stack: [3, 4] queue: [4, 5]
```

The other way to implement it is to write class and implement them using concept of node which shares the same definition as the linked list node. Such implementation can satisfy the $O(1)$ time restriction. For both the stack and queue, we utilize the singly linked list data structure.

Stack and Singly Linked List with top pointer Because in stack, we only need to add or delete item from the rear, using one pointer pointing at the rear item, and the linked list's next is connected to the second toppest item, in a direction from the top to the bottom.

```

1 # stack with linked list
2 '''a<-b<-c<-top'''
3 class Stack:
4     def __init__(self):
5         self.top = None
6         self.size = 0
7
8     # push
9     def push(self, val):
10        node = Node(val)
11        if self.top: # connect top and node
12            node.next = self.top
13        # reset the top pointer
14        self.top = node
15        self.size += 1
16
17    def pop(self):
18        if self.top:
19            val = self.top.val
20            if self.top.next:
21                self.top = self.top.next # reset top
22            else:
23                self.top = None
24            self.size -= 1
25            return val
26
27        else: # no element to pop
28            return None

```

Queue and Singly Linked List with Two Pointers For queue, we need to access the item from each side, therefore we use two pointers pointing at the head and the tail of the singly linked list. And the linking direction is from the head to the tail.

```

1 # queue with linked list
2 '''head->a->b->tail'''
3 class Queue:
4     def __init__(self):
5         self.head = None
6         self.tail = None
7         self.size = 0

```

```

8
9     # push
10    def enqueue(self, val):
11        node = Node(val)
12        if self.head and self.tail: # connect top and node
13            self.tail.next = node
14            self.tail = node
15        else:
16            self.head = self.tail = node
17
18        self.size += 1
19
20    def dequeue(self):
21        if self.head:
22            val = self.head.val
23            if self.head.next:
24                self.head = self.head.next # reset top
25            else:
26                self.head = None
27                self.tail = None
28            self.size -= 1
29            return val
30
31        else: # no element to pop
32            return None

```

Also, Python provide two built-in modules: **Deque** and **Queue** for such purpose. We will detail them in the next section.

9.4.2 Deque: Double-Ended Queue

Deque object is a supplementary container data type from Python **collections** module. It is a generalization of stacks and queues, and the name is short for “double-ended queue”. Deque is optimized for adding/popping items from both ends of the container in $O(1)$. Thus it is preferred over **list** in some cases. To new a deque object, we use **deque([iterable[, maxlen]])**. This returns us a new deque object initialized left-to-right with data from iterable. If maxlen is not specified or is set to None, deque may grow to an arbitrary length. Before implementing it, we learn the functions for **deque class** first in Table 9.6.

In addition to the above, deques support iteration, pickling, **len(d)**, **reversed(d)**, **copy.copy(d)**, **copy.deepcopy(d)**, membership testing with the **in** operator, and subscript references such as **d[-1]**.

Now, we use deque to implement a basic stack and queue, the main methods we need are: **append()**, **appendleft()**, **pop()**, **popleft()**.

```

1 '''Use deque from collections'''
2 from collections import deque
3 q = deque([3, 4])
4 q.append(5)
5 q.popleft()

```

Table 9.6: Common Methods of Deque

Method	Description
append(x)	Add x to the right side of the deque.
appendleft(x)	Add x to the left side of the deque.
pop()	Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.
popleft()	Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.
maxlen	Deque objects also provide one read-only attribute:Maximum size of a deque or None if unbounded.
count(x)	Count the number of deque elements equal to x.
extend(iterable)	Extend the right side of the deque by appending elements from the iterable argument.
extendleft(iterable)	Extend the left side of the deque by appending elements from iterable. Note, the series of left appends results in reversing the order of elements in the iterable argument.
remove(value)	remove the first occurrence of value. If not found, raises a ValueError.
reverse()	Reverse the elements of the deque in-place and then return None.
rotate(n=1)	Rotate the deque n steps to the right. If n is negative, rotate to the left.

```

6
7 s = deque([3, 4])
8 s.append(5)
9 s.pop()

```

Printing out the q and s:

```

1 print('stack:', s, 'queue:', q)
2 stack: deque([3, 4]) queue: deque([4, 5])

```

Deque and Ring Buffer Ring Buffer or Circular Queue is defined as a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. This normally requires us to predefine the maximum size of the queue. To implement a ring buffer, we can use deque as a queue as demonstrated above, and when we initialize the object, set the maxlen. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end.

9.4.3 Python built-in Module: Queue

The **queue module** provides thread-safe implementation of Stack and Queue like data structures. It encompasses three types of queue as shown in Ta-

ble 9.7. In python 3, we use lower case queue, but in Python 2.x it uses Queue, in our book, we learn Python 3.

Table 9.7: Datatypes in Queue Module, maxsize is an integer that sets the upperbound limit on the number of items that can be places in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.

Class	Data Structure
class queue.Queue(maxsize=0)	Constructor for a FIFO queue.
class queue.LifoQueue(maxsize=0)	Constructor for a LIFO queue.
class queue.PriorityQueue(maxsize=0)	Constructor for a priority queue.

Queue objects (Queue, LifoQueue, or PriorityQueue) provide the public methods described below in Table 9.8.

Table 9.8: Methods for Queue's three classes, here we focus on single-thread background.

Class	Data Structure
Queue.put(item[, block[, timeout]])	Put item into the queue.
Queue.get([block[, timeout]])	Remove and return an item from the queue.
Queue.qsize()	Return the approximate size of the queue.
Queue.empty()	Return True if the queue is empty, False otherwise.
Queue.full()	Return True if the queue is full, False otherwise.

Now, using Queue() and LifoQueue() to implement queue and stack respectively is straightforward:

```

1 # python 3
2 import queue
3 # implementing queue
4 q = queue.Queue()
5 for i in range(3, 6):
6     q.put(i)

```

```

1 import queue
2 # implementing stack
3 s = queue.LifoQueue()
4
5 for i in range(3, 6):
6     s.put(i)

```

Now, using the following printing:

```

1 print('stack:', s, ' queue:', q)

```

```
2 stack: <queue.LifoQueue object at 0x000001A4062824A8> queue: <
   queue.Queue object at 0x000001A4062822E8>
```

Instead we print with:

```
1 print('stack: ')
2 while not s.empty():
3     print(s.get(), end=' ')
4 print('\nqueue: ')
5 while not q.empty():
6     print(q.get(), end=' ')
7 stack:
8 5 4 3
9 queue:
10 3 4 5
```

9.4.4 Bonus

Circular Linked List and Circular Queue The circular queue is a linear data structure in which the operation are performed based on FIFO principle and the last position is connected back to the the first position to make a circle. It is also called “Ring Buffer”. Circular Queue can be either implemented with a list or a circular linked list. If we use a list, we initialize our queue with a fixed size with None as value. To find the position of the enqueue(), we use $rear = (rear + 1) \% \text{size}$. Similarly, for dequeue(), we use $front = (front + 1) \% \text{size}$ to find the next front position.

9.4.5 Exercises

Queue and Stack

1. 225. Implement Stack using Queues (easy)
2. 232. Implement Queue using Stacks (easy)
3. 933. Number of Recent Calls (easy)

Queue fits well for buffering problem.

1. 933. Number of Recent Calls (easy)
2. 622. Design Circular Queue (medium)

```
1 Write a class RecentCounter to count recent requests.
2
3 It has only one method: ping(int t), where t represents some
   time in milliseconds.
4
5 Return the number of pings that have been made from 3000
   milliseconds ago until now.
6
```

```

7 Any ping with time in [t - 3000, t] will count, including the
8   current ping.
9 It is guaranteed that every call to ping uses a strictly larger
10  value of t than before.
11
12
13 Example 1:
14
15 Input: inputs = ["RecentCounter","ping","ping","ping","ping"],
16       inputs = [[], [1], [100], [3001], [3002]]
17 Output: [null,1,2,3,3]

```

Analysis: This is a typical buffer problem. If the size is larger than the buffer, then we squeeze out the easiest data. Thus, a queue can be used to save the t and each time, squeeze any time not in the range of [t-3000, t]:

```

1 class RecentCounter:
2
3     def __init__(self):
4         self.ans = collections.deque()
5
6     def ping(self, t):
7         """
8             :type t: int
9             :rtype: int
10            """
11         self.ans.append(t)
12         while self.ans[0] < t - 3000:
13             self.ans.popleft()
14         return len(self.ans)

```

Monotone Queue

1. 84. Largest Rectangle in Histogram
2. 85. Maximal Rectangle
3. 122. Best Time to Buy and Sell Stock II
4. 654. Maximum Binary Tree

Obvious applications:

1. 496. Next Greater Element I
2. 503. Next Greater Element I
3. 121. Best Time to Buy and Sell Stock
4. 84. Largest Rectangle in Histogram

2. 85. Maximal Rectangle
3. 122. Best Time to Buy and Sell Stock II
4. 654. Maximum Binary Tree
5. 42 Trapping Rain Water
6. 739. Daily Temperatures
7. 321. Create Maximum Number

9.5 Hash Table

9.5.1 Implementation

In this section, we practice on the learned concepts and methods by implementing hash set and hash map.

Hash Set Design a HashSet without using any built-in hash table libraries. To be specific, your design should include these functions: (705. Design HashSet)

```
add(value): Insert a value into the HashSet.
contains(value) : Return whether the value exists in the HashSet or not.
remove(value): Remove a value in the HashSet. If the value does not exist in the HashSet, do nothing.
```

For example:

```
MyHashSet hashSet = new MyHashSet();
hashSet.add(1);
hashSet.add(2);
hashSet.contains(1);      // returns true
hashSet.contains(3);      // returns false (not found)
hashSet.add(2);
hashSet.contains(2);      // returns true
hashSet.remove(2);
hashSet.contains(2);      // returns false (already removed)
```

Note: Note: (1) All values will be in the range of [0, 1000000]. (2) The number of operations will be in the range of [1, 10000].

```
1 class MyHashSet:
2
3     def __h(self, k, i):
4         return (k+i) % 10001
5
6     def __init__(self):
7         """
8             Initialize your data structure here.

```

```

9      """
10     self.slots = [None]*10001
11     self.size = 10001
12
13     def add(self, key: 'int') -> 'None':
14         i = 0
15         while i < self.size:
16             k = self._h(key, i)
17             if self.slots[k] == key:
18                 return
19             elif not self.slots[k] or self.slots[k] == -1:
20                 self.slots[k] = key
21                 return
22             i += 1
23     # double size
24     self.slots = self.slots + [None]*self.size
25     self.size *= 2
26     return self.add(key)
27
28
29     def remove(self, key: 'int') -> 'None':
30         i = 0
31         while i < self.size:
32             k = self._h(key, i)
33             if self.slots[k] == key:
34                 self.slots[k] = -1
35                 return
36             elif self.slots[k] == None:
37                 return
38             i += 1
39         return
40
41     def contains(self, key: 'int') -> 'bool':
42         """
43             Returns true if this set contains the specified element
44         """
45         i = 0
46         while i < self.size:
47             k = self._h(key, i)
48             if self.slots[k] == key:
49                 return True
50             elif self.slots[k] == None:
51                 return False
52             i += 1
53         return False

```

Hash Map Design a HashMap without using any built-in hash table libraries. To be specific, your design should include these functions: (706. Design HashMap (easy))

- `put(key, value)` : Insert a (key, value) pair into the HashMap. If the value already exists in the HashMap, update the value.

- `get(key)`: Returns the value to which the specified key is mapped, or `-1` if this map contains no mapping for the key.
- `remove(key)` : Remove the mapping for the value `key` if this map contains the mapping for the `key`.

Example:

```
hashMap = MyHashMap()
hashMap.put(1, 1);
hashMap.put(2, 2);
hashMap.get(1);           // returns 1
hashMap.get(3);           // returns -1 (not found)
hashMap.put(2, 1);         // update the existing value
hashMap.get(2);           // returns 1
hashMap.remove(2);        // remove the mapping for 2
hashMap.get(2);           // returns -1 (not found)
```

```
1 class MyHashMap:
2     def __h(self, k, i):
3         return (k+i) % 10001 # [0, 10001]
4     def __init__(self):
5         """
6             Initialize your data structure here.
7         """
8         self.size = 10002
9         self.slots = [None] * self.size
10
11    def put(self, key: 'int', value: 'int') -> 'None':
12        """
13            value will always be non-negative.
14        """
15        i = 0
16        while i < self.size:
17            k = self.__h(key, i)
18            if not self.slots[k] or self.slots[k][0] in [key,
19                -1]:
20                self.slots[k] = (key, value)
21            return
22            i += 1
23        # double size and try again
24        self.slots = self.slots + [None]* self.size
25        self.size *= 2
26        return self.put(key, value)
27
28    def get(self, key: 'int') -> 'int':
29        """
30            Returns the value to which the specified key is mapped,
31            or -1 if this map contains no mapping for the key
32        """
33        i = 0
34        while i < self.size:
35            k = self.__h(key, i)
```

```

36         if not self.slots[k]:
37             return -1
38         elif self.slots[k][0] == key:
39             return self.slots[k][1]
40         else: # if its deleted keep probing
41             i += 1
42     return -1
43
44
45     def remove(self, key: 'int') -> 'None':
46         """
47             Removes the mapping of the specified value key if this
48             map contains a mapping for the key
49         """
50         i = 0
51         while i < self.size:
52             k = self._h(key, i)
53             if not self.slots[k]:
54                 return
55             elif self.slots[k][0] == key:
56                 self.slots[k] = (-1, None)
57             else: # if its deleted keep probing
58                 i += 1
59     return

```

9.5.2 Python Built-in Data Structures

SET and Dictionary

In Python, we have the standard build-in data structure *dictionary* and *set* using hashtable. For the set classes, they are implemented using dictionaries. Accordingly, the requirements for set elements are the same as those for dictionary keys; namely, that the object defines both `__eq__()` and `__hash__()` methods. A Python built-in function `hash(object =)` is implementing the hashing function and returns an integer value as of the hash value if the object has defined `__eq__()` and `__hash__()` methods. As a result of the fact that `hash()` can only take immutable objects as input key in order to be hashable meaning it must be immutable and comparable (has an `__eq__()` or `__cmp__()` method).

Python 2.X VS Python 3.X In Python 2X, we can use slice to access `keys()` or `items()` of the dictionary. However, in Python 3.X, the same syntax will give us `TypeError: 'dict_keys' object does not support indexing`. Instead, we need to use function `list()` to convert it to list and then slice it. For example:

```

1 # Python 2.x
2 dict.keys()[0]
3

```

```

4 # Python 3.x
5 list(dict.keys())[0]

```

set Data Type Method Description Python Set remove() Removes Element from the Set Python Set add() adds element to a set Python Set copy() Returns Shallow Copy of a Set Python Set clear() remove all elements from a set Python Set difference() Returns Difference of Two Sets Python Set difference_update() Updates Calling Set With Intersection of Sets Python Set discard() Removes an Element from The Set Python Set intersection() Returns Intersection of Two or More Sets Python Set intersection_update() Updates Calling Set With Intersection of Sets Python Set isdisjoint() Checks Disjoint Sets Python Set issubset() Checks if a Set is Subset of Another Set Python Set issuperset() Checks if a Set is Superset of Another Set Python Set pop() Removes an Arbitrary Element Python Set symmetric_difference() Returns Symmetric Difference Python Set symmetric_difference_update() Updates Set With Symmetric Difference Python Set union() Returns Union of Sets Python Set update() Add Elements to The Set.

If we want to put string in set, it should be like this:

```

1 >>> a = set('aardvark')
2 >>>
3 {'d', 'v', 'a', 'r', 'k'}
4 >>> b = {'aardvark'}# or set(['aardvark']), convert a list of
      strings to set
5 >>> b
6 {'aardvark'}
7 #or put a tuple in the set
8 a = set([tuple]) or {(tuple)}

```

Compare also the difference between `and` `set()` with a single word argument.

dict Data Type Method Description clear() Removes all the elements from the dictionary copy() Returns a copy of the dictionary fromkeys() Returns a dictionary with the specified keys and values get() Returns the value of the specified key items() Returns a list containing a tuple for each key value pair keys() Returns a list containing the dictionary's keys pop() Removes the element with the specified key and return value popitem() Removes the last inserted key-value pair setdefault() Returns the value of the specified key. If the key does not exist: insert the key, with the specified value update() Updates the dictionary with the specified key-value pairs values() Returns a list of all the values in the dictionary

See using cases at <https://www.programiz.com/python-programming/dictionary>.

Collection Module

OrderedDict Standard dictionaries are unordered, which means that any time you loop through a dictionary, you will go through every key, but you are not guaranteed to get them in any particular order. The OrderedDict from the collections module is a special type of dictionary that keeps track of the order in which its keys were inserted. Iterating the keys of an ordered-Dict has predictable behavior. This can simplify testing and debugging by making all the code deterministic.

defaultdict Dictionaries are useful for bookkeeping and tracking statistics. One problem is that when we try to add an element, we have no idea if the key is present or not, which requires us to check such condition every time.

```

1 dict = {}
2 key = "counter"
3 if key not in dict:
4     dict[key]=0
5 dict[key] += 1

```

The defaultdict class from the collections module simplifies this process by pre-assigning a default value when a key does not present. For different value type it has different default value, for example, for int, it is 0 as the default value. A defaultdict works exactly like a normal dict, but it is initialized with a function (“default factory”) that takes no arguments and provides the default value for a nonexistent key. Therefore, a defaultdict will never raise a KeyError. Any key that does not exist gets the value returned by the default factory. For example, the following code use a lambda function and provide 'Vanilla' as the default value when a key is not assigned and the second code snippet function as a counter.

```

1 from collections import defaultdict
2 ice_cream = defaultdict(lambda: 'Vanilla')
3 ice_cream[ 'Sarah' ] = 'Chunky Monkey'
4 ice_cream[ 'Abdul' ] = 'Butter Pecan'
5 print ice_cream[ 'Sarah' ]
# Chunky Monkey
6 print ice_cream[ 'Joe' ]
# Vanilla

```

```

1 from collections import defaultdict
2 dict = defaultdict(int) # default value for int is 0
3 dict[ 'counter' ] += 1

```

There include: Time Complexity for Operations Search, Insert, Delete: $O(1)$.

Counter

9.5.3 Exercises

1. 349. Intersection of Two Arrays (easy)
2. 350. Intersection of Two Arrays II (easy)

929. Unique Email Addresses

```

1 Every email consists of a local name and a domain name,
2   separated by the @ sign.
3 For example, in alice@leetcode.com, alice is the local name, and
4   leetcode.com is the domain name.
5 Besides lowercase letters, these emails may contain '.'s or '+'s
6 .
7 If you add periods ('.') between some characters in the local
8   name part of an email address, mail sent there will be
9   forwarded to the same address without dots in the local name.
10  For example, "alice.z@leetcode.com" and "alicez@leetcode.
11  com" forward to the same email address. (Note that this rule
12  does not apply for domain names.)
13 If you add a plus ('+') in the local name, everything after the
14  first plus sign will be ignored. This allows certain emails
15  to be filtered, for example m.y+name@email.com will be
16  forwarded to my@email.com. (Again, this rule does not apply
17  for domain names.)
18 It is possible to use both of these rules at the same time.
19 Given a list of emails, we send one email to each address in the
20  list. How many different addresses actually receive mails?
21 Example 1:
22 Input: ["test.email+alex@leetcode.com", "test.e.mail+bob.
23           cathy@leetcode.com", "testemail+david@lee.tcode.com"]
24 Output: 2
25 Explanation: "testemail@leetcode.com" and "testemail@lee.tcode.
26           com" actually receive mails
27 Note:
28     1 <= emails[i].length <= 100
29     1 <= emails.length <= 100
30     Each emails[i] contains exactly one '@' character.

```

Answer: Use hashmap simply Set of tuple to save the corresponding sending exmail address: local name and domain name:

```

1 class Solution:
2     def numUniqueEmails(self, emails):
3         """
4             :type emails: List[str]

```

```

5     :rtype: int
6     """
7     if not emails:
8         return 0
9     num = 0
10    handledEmails = set()
11    for email in emails:
12        local_name, domain_name = email.split('@')
13        local_name = local_name.split('+')[0]
14        local_name = local_name.replace('.', '')
15        handledEmails.add((local_name, domain_name))
16    return len(handledEmails)

```

9.6 Graph Representations

Graph data structure can be thought of a superset of the array and the linked list, and tree data structures. In this section, we only introduce the presentation and implementation of the graph, but rather defer the searching strategies to the principle part. Searching strategies in the graph makes a starting point in algorithmic problem solving, knowing and analyzing these strategies in details will make an independent chapter as a problem solving principle.

9.6.1 Introduction

Graph representations need to show users full information to the graph itself, $G = (V, E)$, including its vertices, edges, and its weights to distinguish either it is directed or undirected, weighted or unweighted. There are generally four ways: (1) Adjacency Matrix, (2) Adjacency List, (3) Edge List, and (4) optionally, Tree Structure, if the graph is a free tree. Each will be preferred to different situations. An example is shown in Fig 9.3.

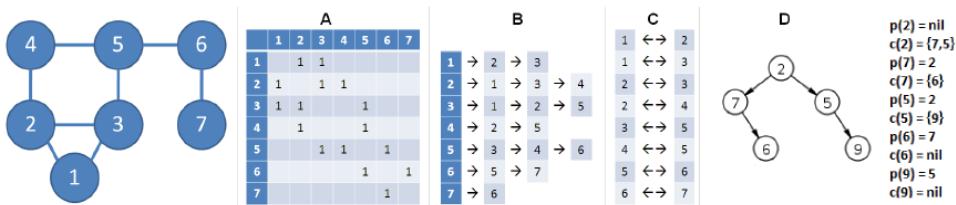


Figure 9.3: Four ways of graph representation, reenumerate it from 0. Redraw the graph

Double Edges in Undirected Graphs In directed graph, the number of edges is denoted as $|E|$. However, for the undirected graph, because one edge (u, v) only means that vertex u and v are connected; we can reach to

v from u and it also works the other way around. To represent undirected graph, we have to double its number of edges shown in the structure; it becomes $2|E|$ in all of our representations.

Adjacency Matrix

An adjacency matrix of a graph is a 2-D matrix of size $|V| \times |V|$: each dimension, row and column, is vertex-indexed. Assume our matrix is `am`, if there is an edge between vertices 3,4, and if its unweighted graph, we mark it by setting `am[3][4]=1`, we do the same for all edges and leaving all other spots in the matrix zero-valued. For undirected graph, it will be a symmetric matrix along the main diagonal as shown in A of Fig. 9.3; the matrix is its own transpose: $am = am^T$. We can choose to store only the entries on and above the diagonal of the matrix, thereby cutting the memory need in half. For unweighted graph, typically our adjacency matrix is zero-and-one valued. For a weighted graph, the adjacency matrix becomes a weight matrix, with $w(i,j)$ to denote the weight of edge (i,j) ; the weight can be both negative or positive or even zero-valued in practice, thus we might want to figure out how to distinguish the non-edge relation from the edge relation when the situation arises.

The Python code that implements the adjacency matrix for the graph in the example is:

```
am = [[0]*7 for _ in range(7)]  
  
# set 8 edges  
am[0][1] = am[1][0] = 1  
am[0][2] = am[2][0] = 1  
am[1][2] = am[2][1] = 1  
am[1][3] = am[3][1] = 1  
am[2][4] = am[4][2] = 1  
am[3][4] = am[4][3] = 1  
am[4][5] = am[5][4] = 1  
am[5][6] = am[6][5] = 1
```

Applications Adjacency matrix usually fits well to the dense graph where the edges are close to $|V|^2$, leaving a small ratio of the matrix be blank and unused. Checking if an edge exists between two vertices takes only $O(1)$. However, an adjacency matrix requires exactly $O(V)$ to enumerate the the neighbors of a vertex v —an operation commonly used in many graph algorithms—even if vertex v only has a few neighbors. Moreover, when the graph is sparse, an adjacency matrix will be both inefficient in the space and iteration cost, a better option is adjacency list.

Adjacency List

An adjacency list is a more compact and space efficient form of graph representation compared with the above adjacency matrix. In adjacency list, we have a list of V vertices which is vertex-indexed, and for each vertex v we store another list of neighboring nodes with their vertex as the value, which can be represented with an array or linked list. For example, with adjacency list as $[[1, 2, 3], [3, 1], [4, 6, 1]]$, node 0 connects to 1,2,3, node 1 connects to 3,1, node 2 connects to 4,6,1.

In Python, We can use a normal 2-d array to represent the adjacent list, for the same graph in the example, it is represented with the following code:

```
a1 = [[] for _ in range(7)]  
  
# set 8 edges  
a1[0] = [1, 2]  
a1[1] = [2, 3]  
a1[2] = [0, 4]  
a1[3] = [1, 4]  
a1[4] = [2, 3, 5]  
a1[5] = [4, 6]  
a1[6] = [5]
```

Applications The upper bound space complexity for adjacency list is $O(|V|^2)$. However, with adjacency list, to check if there is an edge between node u and v , it has to take $O(|V|)$ time complexity with a linear scanning in the list `a1[u]`. If the graph is static, meaning we do not add more vertices but can modify the current edges and its weight, we can use a set or a dictionary Python data type on second dimension of the adjacency list. This change enables $O(1)$ search of an edge just as of in the adjacency matrix.

Edge List

The edge list is a list of edges (one-dimensional), where the index of the list does not relate to vertex and each edge is usually in the form of (starting vertex, ending vertex, weight). We can use either a `list` or a `tuple` to represent an edge. The edge list representation of the example is given:

```
e1 = []  
e1.extend([[0, 1], [1, 0]])  
e1.extend([[0, 2], [2, 0]])  
e1.extend([[1, 2], [2, 1]])  
e1.extend([[1, 3], [3, 1]])  
e1.extend([[3, 4], [4, 3]])  
e1.extend([[2, 4], [4, 2]])  
e1.extend([[4, 5], [5, 4]])  
e1.extend([[5, 6], [6, 5]])
```

Applications Edge list is not widely used as the AM and AL, and usually only be needed in a subroutine of algorithm implementation—such as in Kruskal’s algorithm to fine Minimum Spanning Tree(MST)—where we might need to order the edges by its weight.

Tree Structure

If the connected graph has no cycle and the edges $E = V - 1$, which is essentially a tree. We can choose to represent it either one of the three representations. Optionally, we can use the tree structure is formed as rooted tree with `nodes` which has value and pointers to its children. We will see later how this type of tree is implemented in Python.

9.6.2 Use Dictionary

In the last section, we always use the vertex indexed structure, it works but might not be human-friendly to work with, in practice a vertex always comes with a “name”—such as in the cities system, a vertex should be a city’s name. Another inconvenience is when we have no idea of the total number of vertices, using the index-numbering system requires us to first figure our all vertices and number each, which is an overhead.

To avoid the two inconvenience, we can replace Adjacency list, which is a list of lists with embedded dictionary structure which is a dictionary of dictionaries or sets.

Unweighted Graph For example, we demonstrate how to give a “name” to exemplary graph; we replace 0 with ‘a’, 1 with ‘b’, and the others with {'c', 'd', 'e', 'f', 'g'}. We declare `defaultdict(set)`, the outer list is replaced by the dictionary, and the inner neighboring node list is replaced with a `set` for $O(1)$ access to any edge.

In the demo code, we simply construct this representation from the edge list.

```

1 from collections import defaultdict
2
3 d = defaultdict(set)
4 for v1, v2 in el:
5     d[chr(v1 + ord('a'))].add(chr(v2 + ord('a')))
6 print(d)

```

And the printed graph is as follows:

```
defaultdict(<class 'set'>, {'a': {'b', 'c'}, 'b': {'d', 'c', 'a'}, 'c': {'b', 'e', 'a'}, 'd': {'b', 'e'}, 'e': {'d', 'c', 'f'}, 'f': {'e', 'g'}, 'g': {'f'}})
```

Weighted Graph If we need weights for each edge, we can use two-dimensional dictionary. We use 10 as a weight to all edges just to demonstrate.

```

1 dw = defaultdict(dict)
2 for v1, v2 in el:
3     vn1 = chr(v1 + ord('a'))
4     vn2 = chr(v2 + ord('a'))
5     dw[vn1][vn2] = 10
6 print(dw)
```

We can access the edge and its weight through `dw[v1][v2]`. The output of this structure is given:

```
defaultdict(<class 'dict'>, {'a': {'b': 10, 'c': 10}, 'b': {'a': 10, 'c': 10, 'd': 10}, 'c': {'a': 10, 'b': 10, 'e': 10}, 'd': {'b': 10, 'e': 10}, 'e': {'d': 10, 'c': 10, 'f': 10}, 'f': {'e': 10, 'g': 10}, 'g': {'f': 10}})
```

9.7 Tree Data Structures

In this section, we focus on implementing a **recursive** tree structure, since a free tree just works the same way as of the graph structure. Also, we have already covered the implicit structure of tree in the topic of heap. In this section, we first implement the recursive tree data structure and the construction of a tree. In the next section, we discuss the searching strategies on the tree—tree traversal, including its both recursive and iterative variants.

put an figure here of a binary and n-ary tree.

Because a tree is a hierarchical—here which is represented recursively—structure of a collection of nodes. We define two classes each for the N-ary tree node and the binary tree node. A node is composed of a variable `val` saving the data and children pointers to connect the nodes in the tree.

Binary Tree Node In a binary tree, the children pointers will at most two pointers, which we define as `left` and `right`. The binary tree node is defined as:

```

1 class BinaryNode:
2     def __init__(self, val):
3         self.left = None
4         self.right = None
5         self.val = val
```

N-ary Tree Node For N-ary node, when we initialize the length of the node's children with additional argument `n`.

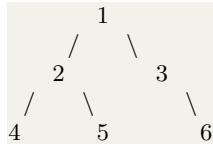
```

1 class NaryNode:
2     def __init__(self, n, val):
3         self.children = [None] * n
```

```
4     self.val = val
```

In this implementation, the children is ordered by each's index in the list. In real practice, there is a lot of flexibility. It is not necessarily to pre-allocate the length of its children, we can start with an empty list [] and just append more nodes to its children list on the fly. Also we can replace the list with a dictionary data type, which might be a better and more space efficient way.

Construct A Tree Now that we have defined the tree node, the process of constructing a tree in the figure will be a series of operations:



```
1 root = BinaryNode(1)
2 left = BinaryNode(2)
3 right = BinaryNode(3)
4 root.left = left
5 root.right = right
6 left.left = BinaryNode(4)
7 left.right = BinaryNode(5)
8 right.right = BinaryNode(6)
```

We see that the above is not convenient in practice. A more practical way is to represent the tree with the heap-like array, which treated the tree as a complete tree. For the above binary tree, because it is not complete in definition, we pad the left child of node 3 with `None` in the list, we would have array [1, 2, 3, 4, 5, `None`, 6]. The root node will have index 0, and given a node with index i , the children nodes of it will be indexed with $n * i + j, j \in [1, \dots, n]$. Thus, a better way to construct the above tree is to start from the array and traverse the list recursively to build up the tree.

We define a recursive function with two arguments: `a`—the input array of nodes and `idx`—indicating the position of the current node in the array. At each recursive call, we construct a `BinaryNode` and set its `left` and `right` child to be a node returned with two recursive call of the same function. Equivalently, we can say these two subprocess—`constructTree(a, 2*idx + 1)` and `constructTree(a, 2*idx + 2)` builds up two subtrees and each is rooted with node $2*idx+1$ and $2*idx+2$ respectively. When there is no items left in the array to be used, it naturally indicates the end of the recursive function and return `None` to indicate its an empty node. We give the following Python code:

```
1 def constructTree(a, idx):
2     """
3         a: input array of nodes
```

```

4     idx: index to indicate the location of the current node
5     '''
6     if idx >= len(a):
7         return None
8     if a[idx]:
9         node = BinaryNode(a[idx])
10        node.left = constructTree(a, 2*idx + 1)
11        node.right = constructTree(a, 2*idx + 2)
12        return node
13    return None

```

Now, we call this function, and pass it with our input array:

```

1 nums = [1, 2, 3, 4, 5, None, 6]
2 root = constructTree(nums, 0)

```

 Please write a recursive function to construct the N-ary tree given in Fig. ???

In the next section, we discuss tree traversal methods, and we will use those methods to print out the tree we just built.

9.7.1 LeetCode Problems

To show the nodes at each level, we use LevelOrder function to print out the tree:

```

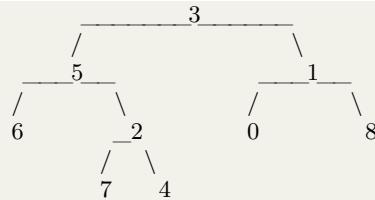
1 def LevelOrder(root):
2     q = [root]
3     while q:
4         new_q = []
5         for n in q:
6             if n is not None:
7                 print(n.val, end=', ')
8                 if n.left:
9                     new_q.append(n.left)
10                if n.right:
11                    new_q.append(n.right)
12        q = new_q
13        print('\n')
14 LevelOrder(root)
15 # output
16 # 1,
17
18 # 2,3,
19
20 # 4,5,None,6,

```

Lowest Common Ancestor. The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself). There will

be two cases in LCA problem which will be demonstrated in the following example.

9.1 Lowest Common Ancestor of a Binary Tree (L236). Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree. Given the following binary tree: root = [3,5,1,6,2,0,8,null,null,7,4]



Example 1:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

Output: 5

Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

Solution: Divide and Conquer. There are two cases for LCA: 1) two nodes each found in different subtree, like example 1. 2) two nodes are in the same subtree like example 2. If we compare the current node with the p and q, if it equals to any of them, return current node in the tree traversal. Therefore in example 1, at node 3, the left return as node 5, and the right return as node 1, thus node 3 is the LCA. In example 2, at node 5, it returns 5, thus for node 3, the right tree would have None as return, thus it makes the only valid return as the final LCA. The time complexity is $O(n)$.

```

1 def lowestCommonAncestor(self, root, p, q):
2     """
3         :type root: TreeNode
4         :type p: TreeNode
5         :type q: TreeNode
6         :rtype: TreeNode
7     """
8     if not root:
9         return None
10    if root == p or root == q:
11        return root # found one valid node (case 1: stop at
12        # 5, case 2:stop at 5)
13    left = self.lowestCommonAncestor(root.left, p, q)
14    right = self.lowestCommonAncestor(root.right, p, q)
15    if left is not None and right is not None: # p, q in
16        the subtree
  
```

```

15     return root
16     if any([left, right]) is not None:
17         return left if left is not None else right
18     return None

```

9.8 Heap

count = Counter(nums)

Heap is a tree based data structure that satisfies *the heap ordering property*. The ordering can be one of two types:

- the min-heap property: the value of each node is greater than or equal (\geq) to the value of its parent, with the minimum-value element at the root.
- the max-heap property: the value of each node is less than or equal to (\leq) the value of its parent, with the maximum-value element at the root.

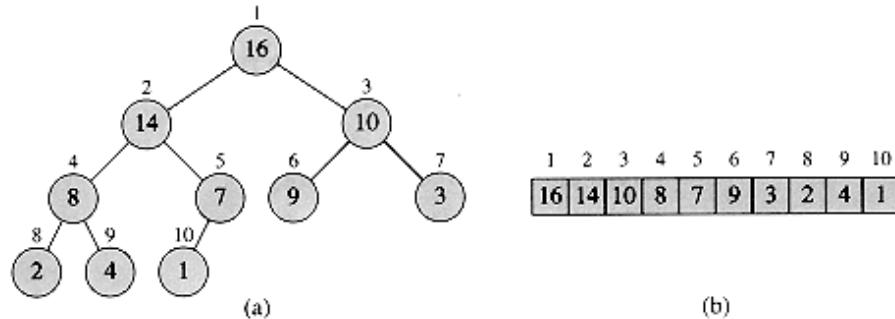


Figure 9.4: Max-heap be visualized with binary tree structure on the left, and be implemented with Array on the right.

Binary Heap A heap is not a sorted structure but can be regarded as partially ordered. The maximum number of children of a node in a heap depends on the type of heap. However, in the more commonly-used heap type, there are at most two children of a node and it's known as a Binary heap. A min-binary heap is shown in Fig. 9.4. Throughout this section the word “heap” will always refer to a min-heap.

Heap is commonly used to implement priority queue that each time the item of the highest priority is popped out – this can be done in $O(\log n)$. As we go through the book, we will find how often priority queue is needed to solve our problems. It can also be used in sorting, such as the heapsort algorithm.

Heap Representation A binary heap is always a complete binary tree that each level is fully filled before starting to fill the next level. Therefore it has a height of $\log n$ given a binary heap with n nodes. A complete binary tree can be uniquely represented by storing its level order traversal in an array. Array representation more space efficient due to the non-existence of the children pointers for each node.

In the array representation, index 0 is skipped for convenience of implementation. Therefore, root locates at index 1. Consider a k-th item of the array, its parent and children relation is:

- its left child is located at $2 * k$ index,
- its right child is located at $2 * k + 1$. index,
- and its parent is located at $k/2$ index (In Python3, use integer division $n//2$).

9.8.1 Basic Implementation

The basic methods of a heap class should include: `push`—push an item into the heap, `pop`—pop out the first item, and `heapify`—convert an arbitrary array into a heap. In this section, we use the heap shown in Fig. 9.5 as our example.

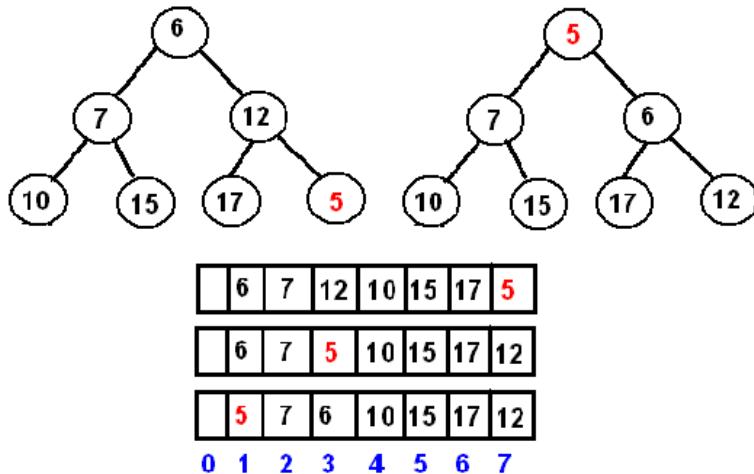


Figure 9.5: A Min-heap.

Push: Percolation Up The new element is initially appended to the end of the heap (as the last element of the array). The heap property is repaired by comparing the added element with its parent and moving the added element up a level (swapping positions with the parent). This process

is called *percolation up*. The comparison is repeated until the parent is larger than or equal to the percolating element. When we push an item in, the item is initially appended to the end of the heap. Assume the new item is the smaller than existing items in the heap, such as 5 in our example, there will be violation of the heap property through the path from the end of the heap to the root. To repair the violation, we traverse through the path and compare the added item with its parent:

- if parent is smaller than the added item, no action needed and the traversal is terminated, e.g. adding item 18 will lead to no action.
- otherwise, swap the item with the parent, and set the node to its parent so that it can keep traverse.

Each step we fix the heap ordering property for a subtree. The time complexity is the same as the height of the complete tree, which is $O(\log n)$.

To generalize the process, a `_float()` function is first implemented which enforce min heap ordering property on the path from a given index to the root.

```

1 def _float(idx, heap):
2     while idx // 2:
3         p = idx // 2
4         # Violation
5         if heap[idx] < heap[p]:
6             heap[idx], heap[p] = heap[p], heap[idx]
7         else:
8             break
9         idx = p
10    return

```

With `_float()`, function `push` is implemented as:

```

1 def push(heap, k):
2     heap.append(k)
3     _float(idx = len(heap) - 1, heap=heap)

```

Pop: Percolation Down When we pop out the item, no matter if it is the root item or any other item in the heap, an empty spot appears at that location. We first move the last item in the heap to this spot, and then start to repair the heap ordering property by comparing the new item at this spot to its children:

- if one of its children has smaller value than this item, swap this item with that child and set the location to that child's location. And then continue.
- otherwise, the process is done.

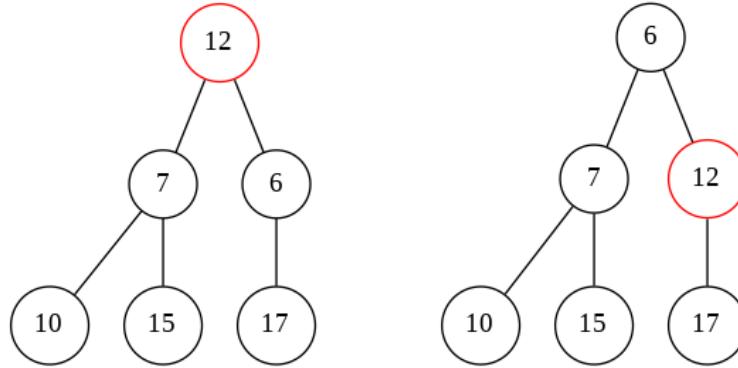


Figure 9.6: Left: delete node 5, and move node 12 to root. Right: 6 is the smallest among 12, 6, and 7, swap node 6 with node 12.

Similarly, this process is called *percolation down*. Same as the insert in the case of complexity, $O(\log n)$. We demonstrate this process with two cases:

- if the item is the root, which is the minimum item 5 in our min-heap example, we move 12 to the root first. Then we compare 12 with its two children, which are 6 and 7. Swap 12 with 6, and continue. The process is shown in Fig. 9.6.
- if the item is any other node instead of root, say node 7 in our example. The process is exactly the same. We move 12 to node 7's position. By comparing 12 with children 10 and 15, 10 and 12 is about to be swapped. With this, the heap ordering property is sustained.

We first use a function `_sink` to implement the percolation down part of the operation.

```

1 def _sink(idx, heap):
2     size = len(heap)
3     while 2 * idx < size:
4         li = 2 * idx
5         ri = li + 1
6         mi = idx
7         if heap[li] < heap[mi]:
8             mi = li
9         if ri < size and heap[ri] < heap[mi]:
10            mi = ri
11        if mi != idx:
12            # swap index with mi
13            heap[idx], heap[mi] = heap[mi], heap[idx]
14        else:
15            break
16        idx = mi

```

The `pop` is implemented as:

```

1 def pop(heap):
2     val = heap[1]
3     # Move the last item into the root position
4     heap[1] = heap.pop()
5     _sink(idx=1, heap=heap)
6     return val

```

Heapify Heapify is a procedure that converts a list to a heap. To heapify a list, we can naively do it through a series of insertion operations through the items in the list, which gives us an upper-bound time complexity : $O(n \log n)$. However, a more efficient way is to treat the given list as a tree and to heapify directly on the list.

To satisfy the heap property, we need to first start from the smallest subtrees, which are leaf nodes. Leaf nodes have no children which satisfy the heap property naturally. Therefore we can jump to the last parent node, which is at position $n//2$ with starting at 1 index. We apply the percolation down process as used in `pop` operation which works forwards comparing the node with its children nodes and applies swapping if the heap property is violated. At the end, the subtree rooted at this particular node obeys the heap ordering property. We then repeat the same process for all parents nodes items in the list in range $[n/2, 1]$ -in reversed order of $[1, n/2]$, which guarantees that the final complete binary tree is a binary heap. This follows a dynamic programming fashion. The leaf nodes $a[n/2 + 1, n]$ are naturally a heap. Then the subarrays are heapified in order of $a[n/2, n]$, $a[n/2 - 1, n], ..., [1, n]$ as we working on nodes $[n/2, 1]$. we first heapify $a[n, n], A[n - 1...n], A[n - 2...n], ..., A[1...n]$. Such process gives us a tighter upper bound which is $O(n)$.

We show how the heapify process is applied on $a = [21, 1, 45, 78, 3, 5]$ in Fig. 9.9.

Implementation-wise, the `heapify` function call `_sink` as its subroutine. The code is shown as:

```

1 def heapify(lst):
2     heap = [None] + lst
3     n = len(lst)
4     for i in range(n//2, 0, -1):
5         _sink(i, heap)
6     return heap

```



Which way is more efficient building a heap from a list?

Using insertion or heapify? What is the efficiency of each method?

The experimental result can be seen in the code.

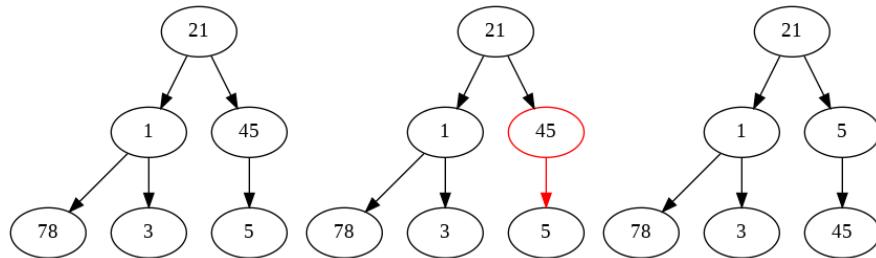


Figure 9.7: Heapify: The last parent node 45.

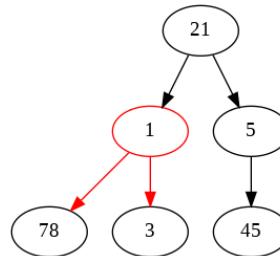


Figure 9.8: Heapify: On node 1

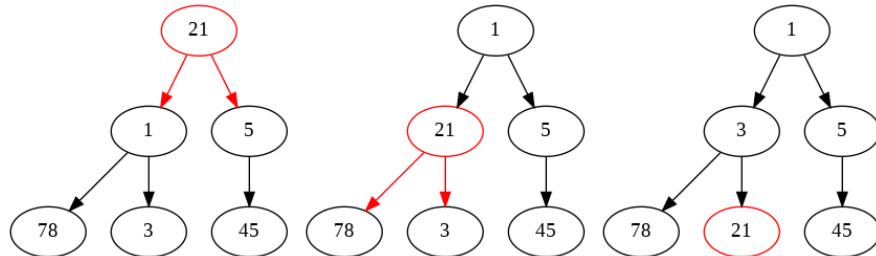


Figure 9.9: Heapify: On node 21.

Try to use the percolation up process to heapify the list.

9.8.2 Python Built-in Library: `heapq`

When we are solving a problem, unless specifically required for implementation, we can always use an existent Python module/package. `heapq` is one of the most frequently used library in problem solving.

`heapq`² is a built-in library in Python that implements heap queue algorithm. `heapq` object implements a minimum binary heap and it provides three main functions: `heappush`, `heappop`, and `heapify` similar to what we have implemented in the last section. The API differs from our last section

²<https://docs.python.org/3.0/library/heapq.html>

in one aspect: it uses zero-based indexing. There are other three functions: `nlargest`, `nsmallest`, and `merge` that come in handy in practice. These functions are listed and described in Table 9.9.

Table 9.9: Methods of `heapq`

Method	Description
<code>heappush(h, x)</code>	Push the <code>x</code> onto the heap, maintaining the heap invariant.
<code>heappop(h)</code>	Pop and return the <i>smallest</i> item from the heap, maintaining the heap invariant. If the heap is empty, <code>IndexError</code> is raised.
<code>heappushpop(h, x)</code>	Push <code>x</code> on the heap, then pop and return the smallest item from the heap. The combined action runs more efficiently than <code>heappush()</code> followed by a separate call to <code>heappop()</code> .
<code>heapify(x)</code>	Transform list <code>x</code> into a heap, in-place, in linear time.
<code>nlargest(k, iterable, key = fun)</code>	This function is used to return the <code>k</code> largest elements from the iterable specified and satisfying the key if mentioned.
<code>nsmallest(k, iterable, key = fun)</code>	This function is used to return the <code>k</code> smallest elements from the iterable specified and satisfying the key if mentioned.
<code>merge(*iterables, key=None, reverse=False)</code>	Merge multiple sorted inputs into a single sorted output. Returns a <i>generator</i> over the sorted values.
<code>heapreplace(h, x)</code>	Pop and return the smallest item from the heap, and also push the new item.

Now, lets see some examples.

Min-Heap Given the exemplary list $a = [21, 1, 45, 78, 3, 5]$, we call the function `heapify()` to convert it to a min-heap.

```
1 from heapq import heappush, heappop, heapify
2 h = [21, 1, 45, 78, 3, 5]
3 heapify(h)
```

The heapified result is $h = [1, 3, 5, 78, 21, 45]$. Let's try `heappop` and `heappush`:

```
1 heappop(h)
2 heappush(h, 15)
```

The print out for `h` is:

```
1 [5, 15, 45, 78, 21]
```

nlargest and nsmallest To get the largest or smallest first n items with these two functions does not require the list to be first heapified with `heapify` because it is built in them.

```
1 from heapq import nlargest, nsmallest
2 h = [21, 1, 45, 78, 3, 5]
3 nl = nlargest(3, h)
4 ns = nsmallest(3, h)
```

The print out for `n1` and `ns` is as:

```
1 [78, 45, 21]
2 [1, 3, 5]
```

Merge Multiple Sorted Arrays Function `merge` merges multiple iterables into a single generator typed output. It assumes all the inputs are sorted. For example:

```
1 from heapq import merge
2 a = [1, 3, 5, 21, 45, 78]
3 b = [2, 4, 8, 16]
4 ab = merge(a, b)
```

The print out of `ab` directly can only give us a generator object with its address in the memory:

```
1 <generator object merge at 0x7fdc93b389e8>
```

We can use list comprehension and iterate through `ab` to save the sorted array in a list:

```
1 ab_lst = [n for n in ab]
```

The print out for `ab_lst` is:

```
1 [1, 2, 3, 4, 5, 8, 16, 21, 45, 78]
```

Max-Heap As we can see the default heap implemented in `heapq` is forcing the heap property of the min-heap. What if we want a max-heap instead? In the library, it does offer us function, but it is intentionally hidden from users. It can be accessed like: `heapq._[function]_max()`. Now, we can heapify a max-heap with function `_heapify_max`.

```
1 from heapq import _heapify_max
2 h = [21, 1, 45, 78, 3, 5]
3 _heapify_max(h)
```

The print out for `h` is:

```
1 [78, 21, 45, 1, 3, 5]
```

Also, in practise, a simple hack for the max-heap is to save data as negative. Whenever we use the data, we convert it to the original value. For example:

```
1 h = [21, 1, 45, 78, 3, 5]
2 h = [-n for n in h]
3 heapify(h)
4 a = -heappop(h)
```

`a` will be 78, as the largest item in the heap.

With Tuple/List or Customized Object as Items for Heap Any object that supports comparison (`_cmp_()`) can be used in heap with `heapq`. When we want our item to include information such as “priority” and “task”, we can either put it in a tuple or a list. `heapq` For example, our item is a list, and the first is the priority and the second denotes the task id.

```

1 heap = [[3, 'a'], [10, 'b'], [5, 'c'], [8, 'd']]
2 heapify(heap)
```

The print out for `heap` is:

```
1 [[3, 'a'], [8, 'd'], [5, 'c'], [10, 'b']]
```

However, if we have multiple tasks that having the same priority, the relative order of these tied tasks can not be sustained. This is because the list items are compared with the whole list as key: it first compare the first item, whenever there is a tie, it compares the next item. For example, when our example has multiple items with 3 as the first value in the list.

```

1 h = [[3, 'e'], [3, 'd'], [10, 'c'], [5, 'b'], [3, 'a']]
2 heapify(h)
```

The printout indicates that the relative ordering of items [3, 'e'], [3, 'd'], [3, 'a'] is not kept:

```
1 [[3, 'a'], [3, 'd'], [10, 'c'], [5, 'b'], [3, 'e']]
```

Keeping the relative order of tasks with same priority is a requirement for *priority queue* abstract data structure. We will see at the next section how priority queue can be implemented with `heapq`.

Modify Items in `heapq` In the heap, we can change the value of any item just as what we can in the list. However, the violation of heap ordering property occurs after the change so that we need a way to fix it. We have the following two private functions to use according to the case of change:

- `_siftdown(heap, startpos, pos)`: `pos` is where the new violation is. `startpos` is till where we want to restore the heap invariant, which is usually set to 0. Because in `_siftdown()` it goes backwards to compare this node with the parents, we can call this function to fix when an item’s value is decreased.
- `_siftup(heap, pos)`: In `_siftup()` items starting from `pos` are compared with their children so that smaller items are sifted up along the way. Thus, we can call this function to fix when an item’s value is increased.

We show one example:

```

1 import heapq
2 heap = [[3, 'a'], [10, 'b'], [5, 'c'], [8, 'd']]
3 heapify(heap)
```

```

4 print(heap)
5
6 heap[0] = [6, 'a']
7 # Increased value
8 heapq._siftup(heap, 0)
9 print(heap)
10 #Decreased Value
11 heap[2] = [3, 'a']
12 heapq._siftdown(heap, 0, 2)
13 print(heap)

```

The printout is:

```

1 [[3, 'a'], [8, 'd'], [5, 'c'], [10, 'b']]
2 [[5, 'c'], [8, 'd'], [6, 'a'], [10, 'b']]
3 [[3, 'a'], [8, 'd'], [5, 'c'], [10, 'b']]]

```

9.9 Priority Queue

A priority queue is an abstract data type(ADT) and an extension of queue with properties:

1. A queue that each item has a priority associated with.
2. In a priority queue, an item with higher priority is served (dequeued) before an item with lower priority.
3. If two items have the same priority, they are served according to their order in the queue.

Priority Queue is commonly seen applied in:

1. CPU Scheduling,
2. Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc.
3. All queue applications where priority is involved.

The properties of priority queue demand sorting stability to our chosen sorting mechanism or data structure. Heap is generally preferred over arrays or linked list to be the underlying data structure for priority queue. In fact, the Python class `PriorityQueue()` from Python module `queue` uses `heapq` under the hood too. We later will see how to implement priority queue with `heapq` and how to use `PriorityQueue()` class for our purpose. In default, the lower the value is, the higher the priority is, making min-heap the underlying data structure.

Implement with heapq Library

The core functions: `heapify()`, `push()`, and `pop()` within `heapq` lib are used in our implementation. In order to implement priority queue, our binary heap needs to have the following features:

- **Sort stability:** when we get two tasks with equal priorities, we return them in the same order as they were originally added. A potential solution is to modify the original 2-element list `[priority, task]` into a 3-element list as `[priority, count, task]`. `list` is preferred because `tuple` does not allow item assignment. The entry `count` indicates the original order of the task in the list, which serves as a tie-breaker so that two tasks with the same priority are returned in the same order as they were added to preserve the sort stability. Also, since no two entry counts are the same so that in the tuple comparison the task will never be directly compared with the other. For example, use the same example as in the last section:

```

1 import itertools
2 counter = itertools.count()
3 h = [[3, 'e'], [3, 'd'], [10, 'c'], [5, 'b'], [3, 'a']]
4 h = [[p, next(counter), t] for p, t in h]

```

The printout for `h` is:

```

1 [[3, 0, 'e'], [3, 1, 'd'], [10, 2, 'c'], [5, 3, 'b'], [3,
4, 'a']]

```

If we `heapify h` will gives us the same order as the original `h`. The relative ordering of ties in the sense of priority is sustained.

- **Remove arbitrary items or update the priority of an item:** In situations such as the priority of a task changes or if a pending task needs to be removed, we have to update or remove an item from the heap. we have seen how to update an item's value in $O(\log n)$ time cost with two functions: `_siftdown()` and `_siftup()` in a heap. However, how to remove an arbitrary item? We could have found and replaced it with the last item in the heap. Then depending on the comparison between the value of the deleted item and the last item, the two mentioned functions can be applied further.

However, there is a more convenient alternative: whenever we “remove” an item, we do not actually remove it but instead simply mark it as “removed”. These “removed” items will eventually be popped out through a normally `pop` operation that comes with heap data structure, and which has the same time cost $O(\log n)$. With this alternative, when we are updating an item, we mark the old item as “removed” and add the new item in the heap. Therefore, with the special

mark method, we are able to implement a heap wherein arbitrary removal and update is supported with just three common functionalities: `heapify`, `heappush`, and `heappop`.

Let's use the same example here. We first remove task 'd' and then update task 'b''s priority to 14. Then we use another list `vh` to get the relative ordering of tasks in the heap according to the priority.

```

1 REMOVED = '<removed-task>'
2 # Remove task 'd'
3 h[1][2] = REMOVED
4 # Update task 'b''s property to 14
5 h[3][2] = REMOVED
6 heappush(h, [14, next(counter), 'b'])
7 vh = []
8 while h:
9     item = heappop(h)
10    if item[2] != REMOVED:
11        vh.append(item)

```

The printout for `vh` is:

```
1 [[3, 0, 'e'], [3, 4, 'a'], [10, 2, 'c'], [14, 5, 'b']]
```

- **Search in constant time:** To search in the heap of an arbitrary item–non-root item and root-item–takes linear time. In practice, tasks should have unique task ids to distinguish from each other, making the usage of a `dictionary` where `task` serves as key and the the 3-element list as value possible (for a list, the value is just a pointer pointing to the starting position of the list). With the dictionary to help search, the time cost is thus decreased to constant. We name this dictionary here as `entry_finder`. Now, with we modify the previous code. The following code shows how to add items into a heap that associates with `entry_finder`:

```

1 # A heap associated with entry_finder
2 counter = itertools.count()
3 entry_finder = {}
4 h = [[3, 'e'], [3, 'd'], [10, 'c'], [5, 'b'], [3, 'a']]
5 heap = []
6 for p, t in h:
7     item = [p, next(counter), t]
8     heap.append(item)
9     entry_finder[t] = item
10 heapify(heap)

```

Then, we execute the remove and update operations with `entry_finder`.

```

1 REMOVED = '<removed-task>'
2 def remove_task(task_id):
3     if task_id in entry_finder:
4         entry_finder[task_id][2] = REMOVED

```

```

5     entry_finder.pop(task_id) # delete from the dictionary
6     return
7
8 # Remove task 'd'
9 remove_task('d')
10 # Updata task 'b' s priority to 14
11 remove_task('b')
12 new_item = [14, next(counter), 'b']
13 heappush(heap, new_item)
14 entry_finder['b'] = new_item

```

In the notebook, we provide a comprehensive class named `PriorityQueue` that implements just what we have discussed in this section.

Implement with `PriorityQueue` class

Class `PriorityQueue()` class has the same member functions as class `Queue()` and `LifoQueue()` which are shown in Table 9.8. Therefore, we skip the introduction. First, we built a queue with:

```

1 from queue import PriorityQueue
2 data = [[3, 'e'], [3, 'd'], [10, 'c'], [5, 'b'], [3, 'a']]
3 pq = PriorityQueue()
4 for d in data:
5     pq.put(d)
6
7 process_order = []
8 while not pq.empty():
9     process_order.append(pq.get())

```

The printout for `process_order` shown as follows indicates how `PriorityQueue` works the same as our `heapq`:

```
1 [[3, 'a'], [3, 'd'], [3, 'e'], [5, 'b'], [10, 'c']]
```

Customized Object If we want the higher the value is the higher priority, we demonstrate how to do so with a customized object with two comparison operators: `<` and `==` in the class with magic functions `__lt__()` and `__eq__()`. The code is as:

```

1 class Job():
2     def __init__(self, priority, task):
3         self.priority = priority
4         self.task = task
5         return
6
7     def __lt__(self, other):
8         try:
9             return self.priority > other.priority
10        except AttributeError:
11            return NotImplemented
12    def __eq__(self, other):

```

```

13     try:
14         return self.priority == other.priority
15     except AttributeError:
16         return NotImplemented

```

Similarly, if we apply the wrapper shown in the second of heapq, we can have a priority queue that is having sort stability, remove and update item, and with constant search time.

In single thread programming, is heapq or PriorityQueue more efficient?

In fact, the PriorityQueue implementation uses heapq under the hood to do all prioritisation work, with the base Queue class providing the locking to make it thread-safe. While heapq module offers no locking, and operates on standard list objects. This makes the heapq module faster; there is no locking overhead. In addition, you are free to use the various heapq functions in different, novel ways, while the PriorityQueue only offers the straight-up queueing functionality.

Hands-on Example

Top K Frequent Elements (L347, medium) Given a non-empty array of integers, return the k most frequent elements.

Example 1:
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]

Example 2:
Input: nums = [1], k = 1
Output: [1]

Analysis: We first use a hashmap to get information as: item and its frequency. Then, the problem becomes obtaining the top k most frequent items in our counter: we can either use sorting or use heap. Our exemplary code here is for the purpose of getting familiar with related Python modules.

- **Counter(). Counter()** has function `most_common(k)` that will return the top k most frequent items. The time complexity is $O(n \log n)$.

```

1 from collections import Counter
2 def topKFrequent(nums, k):
3     return [x for x, _ in Counter(nums).most_common(k)]

```

- **heapq.nlargest()**. The complexity should be better than $O(n \log n)$.

```

1 from collections import Counter
2 import heapq
3 def topKFrequent(nums, k):
4     count = collections.Counter(nums)
5     # Use the value to compare with
6     return heapq.nlargest(k, count.keys(), key=lambda x:
7                           count[x])

```

`key=lambda x: count[x]` can also be replaced with `key=lambda x: count[x].`

- **PriorityQueue():** We put the negative count into the priority queue so that it can perform as a max-heap.

```

1 from queue import PriorityQueue
2 def topKFrequent(self, nums, k):
3     count = Counter(nums)
4     pq = PriorityQueue()
5     for key, c in count.items():
6         pq.put((-c, key))
7     return [pq.get()[1] for i in range(k)]

```

9.10 Bonus

Fibonacci heap With fibonacci heap, `insert()` and `getHighestPriority()` can be implemented in $O(1)$ amortized time and `deleteHighestPriority()` can be implemented in $O(\log n)$ amortized time.

9.11 Exercises

selection with key word: `kth`. These problems can be solved by sorting, using heap, or use quickselect

1. 703. Kth Largest Element in a Stream (easy)
2. 215. Kth Largest Element in an Array (medium)
3. 347. Top K Frequent Elements (medium)
4. 373. Find K Pairs with Smallest Sums (Medium)
5. 378. Kth Smallest Element in a Sorted Matrix (medium)

priority queue or quicksort, quickselect

1. 23. Merge k Sorted Lists (hard)
2. 253. Meeting Rooms II (medium)
3. 621. Task Scheduler (medium)

Part IV

Core Principle: Algorithm Design and Analysis

This part embodies the principle of algorithm design and analysis techniques—the central part of this book.

Before we start, I wanna emphasize that **tree** and **graph** data structure, especially tree, is a great visualization tool to assist us with algorithm design and analysis. Tree is a recursive structure, it can almost used to visualize any recursive based algorithm design or even computing the complexity in which case it is specifically called *recursion tree*.

The next three chapters we introduce the principle of algorithm analysis(chapter 10) and fundamental algorithm design principle—Divide and conquer(Chapter. 13) and Reduce and conquer(Chapter. IV). In Algorithm Analysis, we familiarize ourselves with common concepts and techniques to analyze the performance of algorithms – running time and space complexity. Divide and conquer is a widely used principle in algorithm design, in our book, we dedicate a whole chapter to its sibling design principle – reduce and conquer, which is essentially a superset of optimization design principle—dynamic programming and greedy algorithm—that is further detailed in Chapter. 15 and Chapter. 17.

10

Algorithm Complexity Analysis

When a software program runs on a machine, we genuinely care about the *hardware space* and the *running time* that it takes to complete the execution of the program; space and running time is the cost we need to pay to get the problem solved. The lower the cost, the happier we would be. Thus, **space** and **running time** are two metrics we use to evaluate the performance of programs, or rather say, algorithms.

Now, if I ask you the question, "How to evaluate the performance of algorithms?" Do not go low and tell me, "You just write the code and run it on a computer?" Because here is the reality: (a) These two metrics are mostly possible to vary as using different the physical machine and the programming languages, and (b) The cost will be too high. First, when we are solving a problem, we would always try to come up with many possible solutions—algorithms. Implementing and running all candidates just boost your cost of labor and finance. Second, even at the best case, you only have one candidate, but what if your designated machine can not load the program due to the memory limit, what if your algorithm takes millions of years to run, would you prefer to sit and wait?

With these situation, it is obvious that we need to *predict* algorithm's performance—running time and space—withot implementing or running on a particular machine, and meanwhile the prediction should be independent of the hardwares. In this chapter, we will study the complexity analysis method that strives to enable us such ability. The space complexity is mostly obvious and way easier to obtain compared with its counterpart-time complexity. This decides that in this chapter, the analysis of time complexity will outweigh the pages we spent on space complexity. Before we dive into a plethora of algorithms and data structures, learning the complexity analysis

techniques can help us evaluate each algorithm.

10.1 Introduction

In reality, it is impossible to predict the exact behavior of an algorithm, thus complexity analysis only try to extract the main influencing factors and ignore some trivial details. The complexity analysis is thus only *approximate*, but it works.

What are the main influencing factors? Imagine sorting an array of integers with size 10 and size 10,000,000. The time and space it takes to these two input size will mostly be a huge difference. Thus, the number of items in the *input size* is a straightforward factor. Assume we use n to denote the size of the input, and the complexity analysis will define an expression of the running time as $T(n)$ and the space as $S(n)$.

In complexity analysis, RAM model is based upon, where instructions/operators are executed one after another, without concurrency. Therefore, the running time of algorithm on a particular input can be expressed as counting the number of *operations or “steps”* to run.

What are the difference cases? Yet, when two input instance has exactly the same size, but with different values, such that one array where the input array is already sorted, and the other is totally random, the time it takes to these two cases will possibly vary, depending on the sorting algorithm that you chose. In complexity analysis, *best-case*, *worst-case*, *average-case* complexity analysis is used to differentiate the behavior of the same algorithm applied on different input instance.

1. **Worst-case:** The behavior of the algorithm or an operation of a data structure with respect to the worst possible case of input instance. This gave us a way to measure the upper bound on the running time for any input, which is denoted as O . Knowing it gives us a guarantee that the algorithm will never take any longer.
2. **Average-case:** The expected behavior when the input is randomly drawn from a given distribution. Average case running time is used as an estimate complexity for a normal case. The expected case here offers us asymptotic bound Θ . Computation of average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences. Often it is assumed that all inputs of a given size are equally likely.
3. **Best-case:** The possible best behavior when the input data is arranged in a way, that your algorithms run least amount of time. Best

case analysis can lead us to the lower bound Ω of an algorithm or data structure.

Toy Example: Selection Sort Given a list of integers, sort the item incrementally.

```
For example, given the list A=[10, 3, 9, 2, 8, 7, 9], the sorted
list will be:
A=[2, 3, 7, 8, 9, 9, 10].
```

There are many sorting algorithms, in this case, let us examine the *selection sort*. Given the input array A , and size to be n , we have index $[0, n - 1]$. In selection sort, each time we select the current largest item and swap it with item at its corresponding position in the sorted list, thus dividing the list into two parts: unsorted list on the left and sorted list on the right. For example, at the first pass, we choose 10 from $A[0, n - 1]$ and swap it with $A[n - 1]$, which is 9; at the second pass, we choose the largest item 9 from $A[0, n - 2]$ and swap it with 7 at $A[n - 2]$, and so. Totally, after $n - 1$ passes we will get an incrementally sorted array. More details of selection sort can be found in Chapter 15.

In the implementation, we use `ti` to denote the target position and `li` the index of the largest item which can only get by scanning. We show the Python code:

1	def selectSort(a):	cost	times
2	'''Implement selection sort'''		
3	n = len(a)		
4	for i in range(n - 1): #n-1 passes,		
5	ti = n - 1 - i	c	$n-1$
6	li = 0	c	$n-1$
7	for j in range(n - i):		
8	if a[j] > a[li]:	$c \sum_{i=0}^{n-2}(n-i)$	
9)		
10	li = j	$c \sum_{i=0}^{n-2}(n-i)$	
11)		
12	# swap li and ti		
13	print('swap', a[li], a[ti])		
14	a[ti], a[li] = a[li], a[ti]	c	$n-1$
	print(a)		
	return a		

First, we ignore the distinction between different operation types and treat all alike with a cost of c . In the above code, the line that comes with notations `cost` and `times`—are operations. In line 5, we first point at the target position `ti`. Because of the `for` loop above it, this operation will be called $n - 1$ times. Same for line 6 and 12. For operation in line 8 and 9, the times it operated is denoted as $\sum_{i=0}^{n-2}(n - i)$ due to two nested `for` loops. And the range of j is dependable of the outer loop with i . We get

our running time $T(n)$ by summing up these cost on the variable of i .

$$T(n) = 3c * (n - 1) + \sum_{i=0}^{n-2} 2c(n - i) \quad (10.1)$$

$$= 3c * (n - 1) + 2c(n + (n - 1) + (n - 2) + \dots + 2) \quad (10.2)$$

$$= 3c * (n - 1) + 2c\left(\frac{(n - 1) * (2 + n)}{2}\right) \quad (10.3)$$

$$= cn^2 + cn - 2 + 3cn - 3c$$

$$= cn^2 + 4cn - 3c - 2 \quad (10.2)$$

$$= an^2 + bn + c \quad (10.3)$$

We use three constants a, b, c to rewrite Eq. 10.2 with Eq.10.3.

In the case of sorting, an incrementally sorted array will potentially be the best-cases that takes the least running time and on the other hand decrementally sorted array will be the worst-case. However, in the example of selection sorted array, even if the input is perfect sorted, the algorithm does not consider this case, it still runs $n-1$ passes, each pass it still scans from a fixed size of window to find the largest item (you would only know it is the largest by looking all cases). Thus, in this case, the best-case, worst-case, and average-case all happens to have the same running time shown in Eq. 10.3.

10.2 Asymptotic Notations

Order of Growth and Asymptotic Running Time In Equation 10.3 we end up with three constant a, b, c and two terms with order n^2 and n . When the input is large enough, all the lower order terms, even if with large constant, will become relatively insignificant to the highest term; we thus neglect the lower terms and end up with an^2 . Further, we neglect the constant coefficient a for the same reason. However, we can not say $T(n) = n^2$, because we know mathematically speaking, it is wrong.

Instead, since we are only interested with property of $T(n)$ when n is large enough, we say the relation between the original complexity function $an^2 + bn + c$ is “asymptotically equivalent to” n^2 , which reads “ $T(n)$ is asymptotic to n^2 ” and denoted as $T(n) = an^2 + bn + c \asymp n^2$. From Fig. 10.1, we can visualize that when n is large enough, the term n is trivial compared with n^2 .

In this way, we manage to classify our complexity into a group of families, say, exponential 2^n or polynomial n^2 .

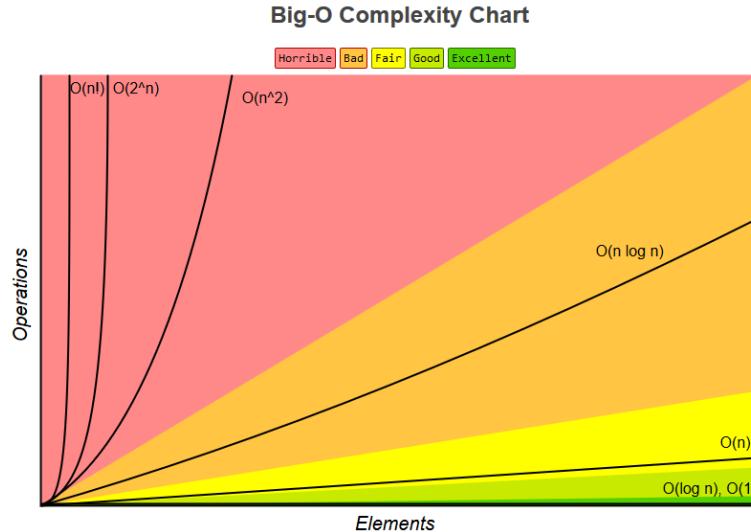


Figure 10.1: Order of Growth of Common Functions

Definition of Asymptotic Notations

We mentioned “asymptotically equivalent” relation, which can be formalized and defined with Θ -Notation as $T(n) = \Theta(n)$, one of the main three asymptotic notations—asymptotically equivalent, smaller, and larger—we will cover in this section.

Θ -Notation For a given function $g(n)$, we define $\Theta(g(n))$ (pronounced as “big theta”) as a set of functions $\Theta(g(n)) = \{f(n)\}$, that each $f(n)$ can be bounded by $g(n)$ by $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ for positive constant c_1 , c_2 and n_0 . We show this relation in Fig. 10.2. Strictly speaking, we would write $f(n) \in \Theta(g(n))$ to indicate that $f(n)$ is just one member of the set of functions that $\Theta(g(n))$ can represent. However, in the field of computer science, we write $f(n) = \Theta(g(n))$ instead.

We say $g(n)$ is an *asymptotically tight bound* of $f(n)$. For example, we can say n^2 is asymptotically tight bound for $2n^2 + 3n + 4$ or $5n^2 + 3n + 4$ or $3n^2$ or any other similar functions. We can denote our running time as $T(n) = \Theta(n^2)$.

O -Notation Further, we define the *asymptotically upper bound* of a set of functions $\{f(n)\}$ as $O(g(n))$ (pronounced as “big oh” of $f(n)$), with $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ for positive constant c and n_0 . We show this relation in Fig. 10.2.

Note that $T(n) = \Theta(g(n))$ implies that $T(n) = O(g(n))$, but not the other way around. With $2n^2 + 3n + 4$ or $5n^2 + 3n + 4$ or $3n^2$, it also be

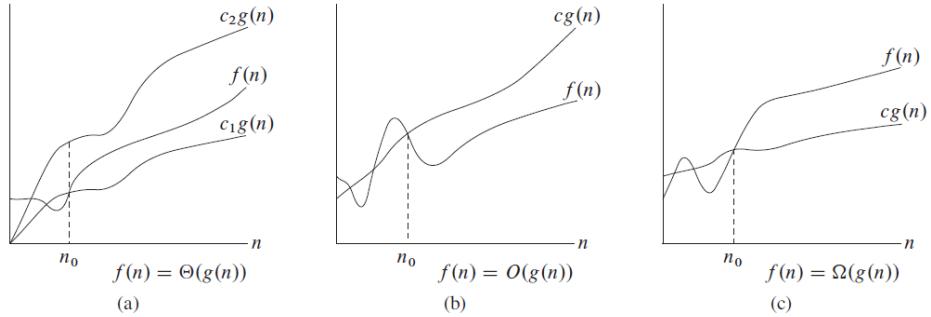


Figure 10.2: Graphical examples for asymptotic notations. Replace $f(n)$ with $T(n)$

denoted as $T(n) = O(n^2)$. Big Oh notation is widely applied in computer science to describe either the running time or the space complexity.

Ω-Notation It provides **asymptotic lower bound** running time. With $T(n) = \Omega(g(n))$ (pronounced as “big omega”) we represent a set of functions that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ for positive constant c and n_0 .

Does it mean that O is worst-case, Θ is the average-case and Ω is the best-case? How does it relate to this three cases.

Properties of Asymptotic Comparisons

We should note that only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, we can have $f(n) = \Theta(g(n))$.

Table 10.1: Analog of Asymptotic Relation

Notation	Similar Relations
$f(n) = \Theta(g(n))$	$f(n) = g(n)$
$f(n) = O(g(n))$	$f(n) \leq g(n)$
$f(n) = \Omega(g(n))$	$f(n) \geq g(n)$

It is fair to denote the relation of $g(n)$ and $f(n)$ to similar relation as between real numbers as shown in Table. 10.1. Thus the properties of real numbers, such as transitivity, reflexivity, symmetry, transpose symmetry all holds for asymptotic notations.

10.3 Practical Guideline

The previous two sections, we introduced the complexity function $T(n)$, how it is influenced by different cases of input instance—worst, average, and best cases, and how that we can use asymptotic notations to focus the complexity only on the dominant term in function $T(n)$. In this section, we would like to provide some practical guideline that arise in real application.

Input Size and Running Time In general, the time taken by an algorithm grows with the size of the input, so it is universal to describe the running time of a program as a function of the size of its input. $f(n)$, with the input size denoted as n .

The notation of **input size** depends on specific problems and data structures. For example, the size of the array can be denoted as integer n , the total numbers of bits when it come to binary notation, and sometimes, if the input is matrix or graph, we need to use two integers such as (m, n) for a two-dimensional matrix or (V, E) for the vertices and edges in a graph.

We use function T to denote the running time. With input size of n , our running time can be denoted as $T(n)$. Given (m, n) , it can be $T(m, n)$.

Worst-case Analysis is Preferred In reality, worst-case input is chosen as our indicator over the best input and average input for: (a) best input is not representative; there is usually an input for the algorithm become trivial; (b) the average-input is sometimes very hard to define and measure; (3) In some cases, the worst-case input is very close to the average and to the observational input; (4)The algorithm with the best efficiency on the worst-case usually achieve the best performance.

Relate Asymptotic Notations to Three Cases of Input Instance

It might seemingly confusing about how the asymptotic notation relates to the three cases of input instance—worst-case, best-case, and average case.

Think about it this way, asymptotic notations apply to any function that it abstract away some lower-term to characterize the property of the function when the input is large or infinite. Therefore, it has nothing to do with these three cases in this way.

However, assume we are trying to characterize the complexity of an algorithm, and we analyzed its best-case and worst case input:

- Worst-case: $T(n) = an^2 + bn + c$, now we can say $T(n) = \Theta(n^2)$, which indicates that $T(n) = \Omega(n^2)$ and $T(n) = O(n^2)$.
- Best-case: $T(n) = an$, we can say $T(n) = \Theta(n)$, which indicates that $T(n) = \Omega(n)$ and $T(n) = O(n)$.

In order to describe the complexity of our algorithm in general; put aside the particular input instance. Such as the the average case analysis, which is typically hard to “average” between different input, we can come up with an estimation, and safely say for the time complexity in general is $an \leq T(n) \leq an^2 + bn + c$. This can be further expanded as:

$$c_1n \leq an \leq T(n) \leq an^2 + bn + c \leq c_2n^2 \quad (10.4)$$

Equivalently, we are safe to characterize a lower-bound based on best-case and an upper-bound based on the worst-case, thus we say the time complexity of our algorithm as $T(n) = \Omega(n), T(n) = O(n^2)$.

Big Oh is a Popular Notation to Complexity Analysis As we have concluded that the worst-case analysis is both easy to get and good indicator of the overall complexity. Big Oh as the absolute upper bound of the worst-case would also indicate the upper bound of the algorithm in general.

Even if we can get a tight bound for the algorithm as in the case of selection sort, it is always right to say that its an upper bound because $\Theta(g(n))$ is a subset of $O(g(n))$. This is like, we know dog is categorized as canine, and canine is in the type of mammal, thus, we are right to say that dog is a species of mammal.

10.4 Time Recurrence Relation

We have studied recurrence relation throughly in Chapter. II. How does it relate to complexity analysis? We can represent either recursive function or iterative function with time recurrence relation. Therefore, the complexity analysis can be done in two steps: (1) get the recurrence relation and (2) solve the recurrence relation.

- For recursive function, this representation is natural. For example, in the merge sort, it can be easily represented as $T(n) = 2T(n/2) + O(n)$, that each step it divides a problem of size n into two subproblems each with half size, and the cost to combine the solution of these two subproblems will be at most n , that is why we add $O(n)$.
- A time recurrence relation can be easily applied on iterative program too. Say, in the simple task where we try to search a target in a list array, we can write a recurrence relation function to it as $T(n) = T(n - 1) + 1$. Because, in the scanning process, one move reduce the problem to a smaller size, and the case of it is 1. Using the asymptotic notation, we can further write it as $T(n) = T(n - 1) + O(1)$. Solving this recurrence relation straightforwardly through iteration method, we can have $T(n) = O(n)$.

As in the chapter. ??, there are generally two ways of reducing a problem: divide and conquer and Reduce by Constant size, which is actually a non-homogenous recurrence relation.

In Chapter. II, we showed how to solve linear recurrence relation and get absolute answer, it was seemingly complex and terrifying. Good news, as complexity analysis is about estimating the cost, so we can loose ourselves a bit and sometimes a lower/upper bound is good enough, and the base case will almost always be $O(1) = 1$.

10.4.1 General Methods to Solve Recurrence Relation

We have shown in Chapter. II there are iterative method and mathematical induction as general methods to try to solve an easy recurrence relation. We demonstrate how these two methods can be used in solving time recurrence relations first. Additionally, we introduce recursion tree method.

Iterative Method

The most straightforward method for solving recurrence relation no matter its linear or non-linear is the *iterative method*. Iterative method is a technique or procedure in computational mathematics that it iteratively replace/substitute each a_n with its recurrence relation $\Psi(n, a_{n-1}, a_{n-2}, \dots, a_{n-k})$ till all items “disappear” other than the initial values. Iterative method is also called substitution method.

We demonstrate iteration with a simple non-overlapping recursion.

$$T(n) = T(n/2) + O(1) \quad (10.5)$$

$$= T(n/2^2) + O(1) + O(1)$$

$$= T(n/2^3) + 3O(1)$$

$$= \dots$$

$$= T(1) + kO(1) \quad (10.6)$$

We have $\frac{n}{2^k} = 1$, we solve this equation and will get $k = \log_2 n$. Most likely $T(1) = O(1)$ will be the initial condition, we replace this, and we get $T(n) = O(\log_2 n)$.

However, when we try to apply iteration on the third recursion: $T(n) = 3T(n/4) + O(n)$. It might be tempting to assume that $T(n) = O(n \log n)$

due to the fact that $T(n) = 2T(n/2) + O(n)$ leads to this time complexity.

$$T(n) = 3T(n/4) + O(n) \quad (10.7)$$

$$\begin{aligned} &= 3(3T(n/4^2) + n/4) + n = 3^2T(n/4^2) + n(1 + 3/4) \\ &= 3^2(3T(n/4^3) + n/4^2) + n(1 + 3/4) = 3^3T(n/4^3) + n(1 + 3/4 + 3/4^2) \end{aligned} \quad (10.8)$$

$$= \dots \quad (10.9)$$

$$= 3^kT(n/4^k) + n \sum_{i=0}^{k-1} \left(\frac{3}{4}\right)^i \quad (10.10)$$

Recursion Tree

Since the term of $T(n)$ grows, the iteration can look messy. We can use recursion tree to better visualize the process of iteration. In a recursive tree, each node represents the value of a single subproblem, and a leaf would be a subproblem. As a start, we expand $T(n)$ as a node with value n as root, and it would have three children each represents a subproblem $T(n/4)$. We further do the same with each leaf node, until the subproblem is trivial and be a base case. In practice, we just need to draw a few layers to find the rule. The cost will be the sum of costs of all layers. The process can be seen in Fig. 10.3. In this case, it is the base case $T(1)$. Through the expansion

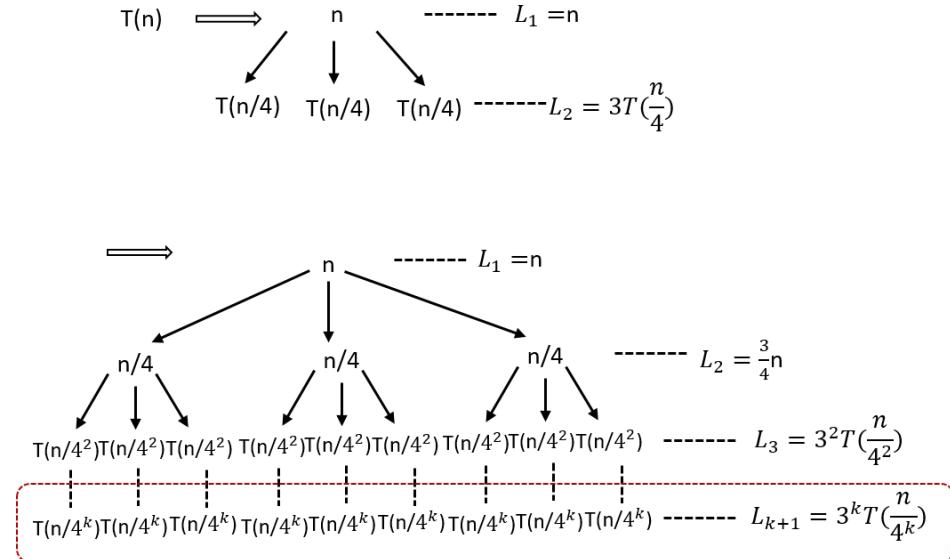


Figure 10.3: The process to construct a recursive tree for $T(n) = 3T([n/4]) + O(n)$. There are totally $k+1$ levels. Use a better figure.

with iteration and recursion tree, our time complexity function becomes:

$$T(n) = \sum_{i=1}^k L_i + L_{k+1} \quad (10.11)$$

$$= n \sum_{i=1}^k (3/4)^{i-1} + 3^k T(n/4^k) \quad (10.12)$$

In the process, we can see that Eq. 10.13 and Eq. 10.7 are the same. Because $T(n/4^k) = T(1) = 1$, we have $k = \log_4 n$.

$$T(n) \leq n \sum_{i=1}^{\infty} (3/4)^{i-1} + 3^k T(n/4^k) \quad (10.13)$$

$$\leq 1/(1 - 3/4)n + 3^{\log_4 n} T(1) = 4n + n^{\log_4 3} \leq 5n \quad (10.14)$$

$$= O(n) \quad (10.15)$$

Mathematical Induction

Mathematical induction is a mathematical proof technique, and is essentially used to prove that a property $P(n)$ holds for every natural number n , i.e. for $n = 0, 1, 2, 3$, and so on. Therefore, in order to use induction, we need to make a *guess* of the closed-form solution for a_n . Induction requires two cases to be proved.

1. *Base case*: proves that the property holds for the number 0.
2. *Induction step*: proves that, if the property holds for one natural number n , then it holds for the next natural number $n + 1$.

For $T(n) = 2 \times T(n - 1) + 1$, $T_0 = 0$, we can have the following result by expanding $T(i), i \in [0, 7]$.

n	0	1	2	3	4	5	6	7
T_n	0	3	7	15	31	63	127	

It is not hard that we find the rule and guess $T(n) = 2^n - 1$. Now, we prove this equation by induction:

1. Show that the basis is true: $T(0) = 2^0 - 1 = 0$.
2. Assume it holds true for $T(n - 1)$. By induction, we get

$$T(n) = 2T(n - 1) + 1 \quad (10.16)$$

$$= 2(2^{n-1} - 1) + 1 \quad (10.17)$$

$$= 2^n - 1 \quad (10.18)$$

Now we show that the induction step holds true too.



Solve $T(n) = T(n/2) + O(1)$ and $T(2n) \leq 2T(n) + 2n - 1$, $T(2) = 1$.

10.4.2 Solve Divide-and-Conquer Recurrence Relations

All the previous recurrence relation, either homogeneous or non-homogeneous, they fall into the bucket of *decrease and conquer* (maybe not right), and either is yet another type of recursion—Divide and Conquer. Same here, we ignore how we get such recurrence but focus on how to solve it.

We write our divide and conquer recurrence relation using the time complexity function, there are two types as shown in Eq.10.19(n are divided equally) and E1.10.20(n are divided unequally):

$$T(n) = aT(n/b) + f(n) \quad (10.19)$$

where $a \leq 1, b > 1$, and $f(n)$ is a given function, which usually has $f(n) = cn^k$.

$$T(n) = \sum_{i=1}^k a_i T(n/b_i) + f(n) \quad (10.20)$$

Considering that the first type is much more commonly seen than the other, we only learn how to solve the first type; in fact, at least, I assume you that within this book, the second type will never appear.

Sit and Deduct For simplicity, we assume $n = b^m$, so that n/b is always integer. First, let us use the iterative method, and expand Eq. 10.19 up till n/b^m times so that $T(n)$ become $T(1)$:

$$T(n) = aT(n/b) + cn^k \quad (10.21)$$

$$= a(aT(n/b^2) + c(n/b)^k) + cn^k \quad (10.22)$$

$$= a(a(T(n/b^3) + c(n/b^2)^k) + c(n/b)^k) + cn^k \quad (10.23)$$

$$\vdots \quad (10.24)$$

$$= a(a(\dots T(n/b^m) + c(n/b^{m-1})^k) + \dots) + cn^k \quad (10.25)$$

$$= a(a(\dots T(1) + cb^k) + \dots) + cn^k \quad (10.26)$$

Now, assume $T(1) = c$ for simplicity and for getting rid of this constant part in our sequence. Then,

$$T(n) = ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \dots + cb^{mk}, \quad (10.27)$$

which implies that

$$T(n) = c \sum_{i=0}^m a^{m-i} b^{ik} \quad (10.28)$$

$$= ca^m \sum_{i=0}^m \left(\frac{b^k}{a}\right)^i \quad (10.29)$$

So far, we get a geometric series, which is a good sign to get the closed-form expression. We first summarize all possible substitutions that will help our further analysis.

$$f(n) = cn^k \quad (10.30)$$

$$n = b^m \quad (10.31)$$

$$\rightarrow \quad (10.32)$$

$$m = \log_b n \quad (10.33)$$

$$f(n) = cb^{mk} \quad (10.34)$$

$$a^m = a^{\log_b n} = n^{\log_b a} \quad (10.35)$$

Depending on the relation between a and b^k , there are three cases:

1. $b^k < a$: In this case, $\frac{b^k}{a} < 1$, so the geometric series converges to a constant even if m goes to infinity. Then, we have an upper bound for $T(n)$, $T(n) < ca^m$, which converts to $T(n) = O(a^m)$. According to Eq. 10.35, we further get:

$$T(n) = O(n^{\log_b a}) \quad (10.36)$$

2. $b^k = a$: With $\frac{b^k}{a} = 1$, $T(n) = O(a^m m)$. With Eq. 10.35 and Eq. 10.33, our upper bound is:

$$T(n) = O(n^{\log_b a} \log_b n) \quad (10.37)$$

3. $b^k > a$: In this case, we denote $\frac{b^k}{a} = d$ (d is a constant and $d > 1$). Use the standard formula for summing over a geometric series:

$$T(n) = ca^m \frac{d^{m+1} - 1}{d - 1} = O(a^m \frac{d^{m+1} - 1}{d - 1}) \quad (10.38)$$

$$= O(b^{mk}) = O(n^k) = O(f(n)) \quad (10.39)$$

Master Method

Comparison between b^k and a equals to the comparison between b^{km} between a^m . From the above substitution, it further equals to compare $f(n)$ to

$n^{\log_b a}$. This is when master method kicks in and we will see how it helps us to apply these three cases into real situation.

Compare $f(n)/c = n^k$ with $n^{\log_b a}$. Intuitively, the larger of the two functions would dominate the solution to the recurrence. Now, we rephrase the three cases using the master method for the easiness of memorization.

1. If $n^k < n^{\log_b a}$, or say *polynomially smaller* than by a factor of n^ϵ for some constant $\epsilon > 0$, we have:

$$T(n) = O(n^{\log_b a}) \quad (10.40)$$

2. If $n^k > n^{\log_b a}$, similarly, we need it to be polynomially larger than a factor of n^ϵ for some constant $\epsilon > 0$, we have:

$$T(n) = O(f(n)) \quad (10.41)$$

3. If $n^k = n^{\log_b a}$, then:

$$T(n) = O(n^{\log_b a} \log_b n) \quad (10.42)$$

10.4.3 Hands-on Example: Insertion Sort

In this section, we are expecting to see example that has different asymptotic bound as the input differs; where we focus more on the worst-case and average-case analysis. Along the analysis of complexity, we will also see how asymptotic notation can be used in equations or inequalities to assist the process.

Because most of the time, the average-case running time will be asymptotically equal to the worst-case, thus we do not really try to analyze it at the first place. In the case of best-case, it would only matter if you know your application context fits right in, otherwise, it will be trivial and non-helpful in the comparison of multiple algorithms. We will see example below.

Insertion Sort: Worst-case and Best-case There is another sorting algorithm—insertion sort—it sets aside another array S to save the sorted items. At first, we can put the first item in which itself is already sorted. At the second pass, we put $A[1]$ into the right position in S . Until the last item is handled, we return the sorted list. The code is:

```

1 def insertionSort(a):
2     '''implement insertion sort'''
3     if not a or len(a) == 1:
4         return a
5     n = len(a)
6     sl = [a[0]] + [None] * (n-1) # sorted list
7     for i in range(1, n): # items to be inserted into the sorted
8         key = a[i]

```

```

9     j = i-1
10
11    while j >= 0 and s1[j] > key: # compare key from the last
12        s1[j+1] = s1[j] # shift a[j] backward
13        j == 1
14        s1[j+1] = key
15        print(s1)
16    return s1

```

For the first **for** loop in line 7, it will sure has $n - 1$ passes. However, for the inner **while** loop, the real times of execution of statement in line 12 and 13 depends on the state between **s1** and **key**. If we try to sort the input array **a** incrementally such that $A=[2, 3, 7, 8, 9, 9, 10]$, and if the input array is already sorted, then there will be no items in the sorted list can be larger than our key which result only the execution of line 14. This is the best case, we can denote the running time of the **while** loop by $\Omega(1)$ because it has constant running time at its best case. However, if the input array is a reversed as the desired sorting, which means it is decreasing sorted such as $A=[10, 9, 9, 9, 7, 3, 2]$, then the inner **while** loop will has $n - i$, we denote it by $O(n)$. We can denote our running time equation as:

$$\begin{aligned} T(n) &= T(n - 1) + O(n) \\ &= O(n^2) \end{aligned} \tag{10.43}$$

And,

$$\begin{aligned} T(n) &= T(n - 1) + \Omega(1) \\ &= \Omega(n) \end{aligned} \tag{10.44}$$

Using simple iteration, we can solve the math formula and have the asymptotic upper bound and lower bound for the time complexity of insertion sort.

For the average case, we can assume that each time, we need half time of comparison of $n - i$, we can have the following equation:

$$\begin{aligned} T(n) &= T(n - 1) + \Theta(n/2) \\ &= T(n - 2) + \Theta(\frac{n}{2} + \frac{n-1}{2}) \\ &= \Theta(n^2) \end{aligned} \tag{10.45}$$

For algorithm that is stable in complexity, we conventionally analyze its average performance, and it is better to use Θ -notation in the running time equation and give the asymptotic tight bound like in the selection sort. For algorithm such as insertion sort, whose complexity varies as the input data distribution we conventionally analyze its worst-case and use O -notation.

10.5 *Amortized Analysis

There are two different ways to evaluate an algorithm/data structure:

1. Consider each operation separately: one that looks each operation incurred in the algorithm/data structure separately and offers worst-case running time O and average running time Θ for each operation. For the whole algorithm, it sums up on these two cases by how many times each operation is incurred.
2. Amortized among a sequence of (related) operations: Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

Amortized analysis does not purely look each operation on a given data structure separately, it averages time required to perform a sequence of different data structure operations over all performed operations. With amortized analysis, we might see that even though one single operation might be expensive, the amortized cost of this operation on all operations is small. Different from average-case analysis, probability will not be applied. From the example later we will see that amortized analysis views the data structure in applicable scenario, to complete this tasks, what is the average cost of each operation, and it is achievable given any input. Therefore, the same time complexity, say $O(f(n))$, worst-case > amortized > average.

There are three types of amortized analysis:

1. Aggregate Analysis:
2. Accounting Method:
3. Potential method:

10.6 Space Complexity

The analysis of space complexity is more straightforward, given that we are essentially the ones who allocate space for the application. We simply link it to the size of items in the data structures. The only obscure is with *recursive program* which takes space from stack but is hidden from the users by the programming language compiler or interpreter. The recursive program can be represented as a recursive tree, the maximum stack space it needs is decided by the height of the recursive tree, thus $O(h)$, given h as the height.

Space and Time Trade-off In the field of algorithm design, we can usually trade space for time efficiency or trade time for space efficiency. For example, if you put your algorithm on a backend server, we need to response the request of users, then decrease the response time if especially useful here. Normally we want to decrease the time complexity by sacrificing more space if the extra space is not a problem for the physical machine. But in some cases, decrease the time complexity is more important and needed, thus we need might go for alternative algorithms that uses less space but might with more time complexity.

10.7 Summary

For your convenience, we provide a table that shows the frequent used recurrence equations' time complexity.

Equation	Time	Space	Examples
$T(n) = 2^*T(n/2) + O(n)$	$O(n\log n)$	$O(\log n)$	quick_sort
$T(n) = 2^*T(n/2) + O(n)$	$O(n\log n)$	$O(n + \log n)$	merge_sort
$T(n) = T(n/2) + O(1)$	$O(\log n)$	$O(\log n)$	Binary search
$T(n) = 2^*T(n/2) + O(1)$	$O(n)$	$O(\log n)$	Binary tree traversal
$T(n) = T(n-1) + O(1)$	$O(n)$	$O(n)$	Binary tree traversal
$T(n) = T(n-1) + O(n)$	$O(n^2)$	$O(n)$	quick_sort(worst case)
$T(n) = n * T(n-1)$	$O(n!)$	$O(n)$	permutation
$T(n) = T(n-1)+T(n-2)+...+T(1)$	$O(2^n)$	$O(n)$	combination

Figure 10.4: The cheat sheet for time and space complexity with recurrence function. If $T(n) = T(n-1)+T(n-2)+...+T(1)+O(n-1) = 3^n$. They are called factorial, exponential, quadratic, linearithmic, linear, logarithmic, constant.

10.8 Exercises

10.8.1 Knowledge Check

1. Use iteration and recursion tree to get the time complexity of $T(n) = T(n/3) + 2T(2n/3) + O(n)$.
2. Get the time complexity of $T(n) = 2T(n/2) + O(n^2)$.
3. $T(n) = T(n - 1) + T(n - 2) + T(n - 3) + \dots + T(1) + O(1)$.

11

Search Strategies

Our standing at graph algorithms:

1. Search Strategies (Current)
2. Combinatorial Search(Chapter)
3. Advanced Graph Algorithm(Current)
4. Graph Problem Patterns(Future Chapter)

Searching¹ is one of the most effective tools in algorithms. We have seen them being widely applied in the field of artificial intelligence to offer either exact or approximate solutions for complex problems such as puzzles, games, routing, scheduling, motion planning, navigation, and so on. On the spectrum of discrete problems, nearly every single one can be modeled as a searching problem together with enumerative combinatorics and **optimizations**. The searching solutions serve as either naive baselines or even as the only existing solutions for some problems. Understanding common searching strategies as the main goal of this chapter along with the search space of the problem lays the foundation of problem analysis and solving, it is just indescribably **powerful** and **important**!

11.1 Introduction

Linear, tree-like data structures, they are all subsets of graphs, making graph searching universal to all searching algorithms. There are many searching

¹https://en.wikipedia.org/wiki/Category:Search_algorithms

strategies, and we only focus on a few decided upon the completeness of an algorithm—being absolutely sure to find an answer if there is one.

Searching algorithms can be categorized into the following two types depending on if the domain knowledge is used to guide selection of the best path while searching:

1. Uninformed Search: This set of searching strategies normally are handled with basic and obvious problem definition and are not guided by estimation of how optimistic a certain node is. The basic algorithms include: Depth-first-Search(DFS), Breadth-first Search(BFS), Bidirectional Search, Uniform-cost Search, Iterative deepening search, and so on. We choose to cover the first four.
2. Informed(Heuristic) Search: This set of searching strategies on the other hand, use additional domain-specific information to find a *heuristic function* which estimates the cost of a solution from a node. Heuristics means “serving to aid discovery”. Common algorithms seen here include: Best-first Search, Greedy Best-first Search, A^* Search. And we only introduce Best-first Search.

Following this introductory chapter, in Chapter Combinatorial Search, we introduce combinatorial problems and its search space, and how to prune the search space to search more efficiently.

Because the search space of a problem can either be of linear or tree structure—an implicit free tree, which makes the graph search a “big deal” in practice of problem solving. Compared with reduce and conquer, searching algorithms treat states and actions atomically: they do not consider any internal/optimal structure they might posses. We recap the **linear search** given its easiness and that we have already learned how to search in multiple linear data structures.

Linear Search As the naive and baseline approach compared with other searching algorithms, linear search, a.k.a sequential search, simply traverse the linear data structures sequentially and checking items until a target is found. It consists of a `for/while` loop, which gives as $O(n)$ as time complexity, and no extra space needed. For example, we search on list A to find a target t :

```

1 def linearSearch(A, t): #A is the array , and t is the target
2     for i,v in enumerate(A):
3         if A[i] == t:
4             return i
5     return -1

```

Linear Search is rarely used practically due to its lack of efficiency compared with other searching methods such as hashmap and binary search that we will learn soon.

Searching in Un-linear Space For the un-linear data structure, or search space comes from combinatorics, they are generally be a graph and sometimes be a rooted tree. Because mostly the search space forms a search tree, we introduce searching strategies on a search tree first, and then we specifically explore searching in a tree, recursive tree traversal, and search in a graph.

Generatics of Search Strategies

Assume we know our state space, searching or state-space search is the process of searching through a state space for a solution by making explicit a sufficient portion of an implicit state-space graph, in the form of a search tree, to include a goal node.

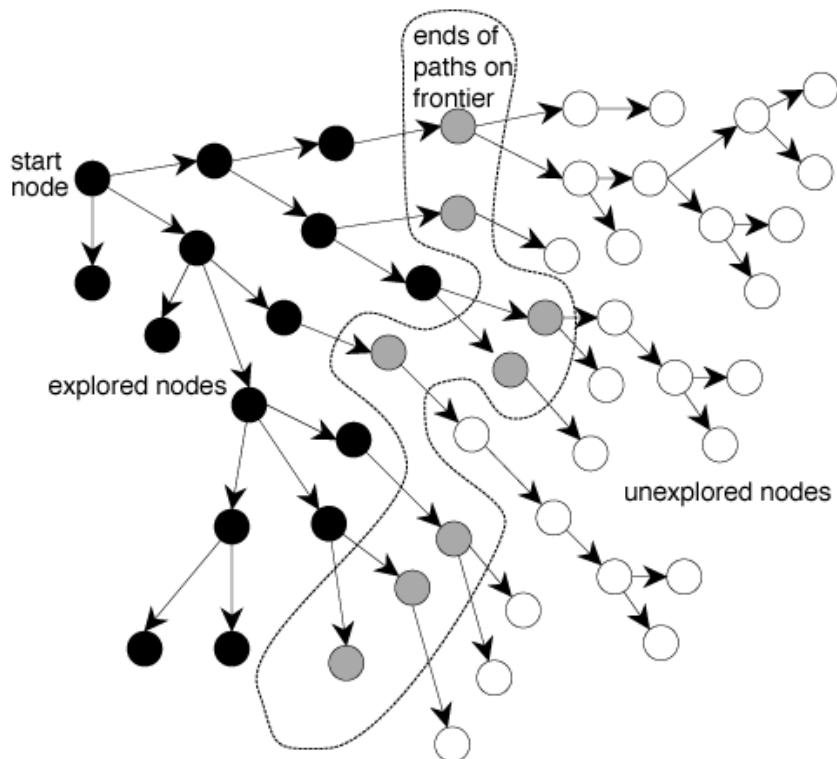


Figure 11.1: Graph Searching

Nodes in Searching Process In the searching process, nodes in the targeting data structure can be categorized into three sets as shown in Fig.11.1 and we distinguish the state of a node—which set they are at with a color each.

- Unexplored set–WHITE: initially all nodes in the graph are in the unexplored set, and we assign WHITE color. Nodes in this set have not yet been visited.
- Frontier set–GRAY: nodes which themselves have been just discovered/visited and they are put into the *frontier* set, waiting to be expanded; that is to say their children or adjacent nodes (through outgoing edges) are about to be discovered and have not all been visited—not all being found in the frontier set yet. This is an intermediate state between WHITE and BLACK, which is ongoing, visiting but not yet completed. Gray vertex might have adjacent vertices of all three possible states.
- Explored set–BLACK: nodes have been fully explored after being in the frontier set; that is to say none of their children is not explored and being in the unexplored set. For black vertex, all vertices adjacent to them are nonwhite.

All searching strategies follow the general tree search algorithm:

1. At first, put the state node in the frontier set.

```
1 frontier = {S}
```

2. Loop through the frontier set, if it is empty then searching terminates. Otherwise, pick a node n from frontier set:
 - (a) If n is a goal node, then return solution
 - (b) Otherwise, generate all of n 's successor nodes and add them all to frontier set.
 - (c) Remove n from frontier set.

Search process constructs a *search tree* where the root is the start state. Loops in graph may cause the search tree to be infinite even if the state space is small. In this section, we only use either acyclic graph or tree for demonstrating the general search methods. In acyclic graph, there might exist multiple paths from source to a target. For example, the example shown in Fig. ?? has multiple paths from to. Further in graph search section, we discuss how to handle cycles and explain single-path graph search. Changing the ordering in the frontier set leads to different search strategies.

11.2 Uninformed Search Strategies

Through this section, we use Fig. 11.2 as our exemplary graph to search on. The data structure to represent the graph is as:

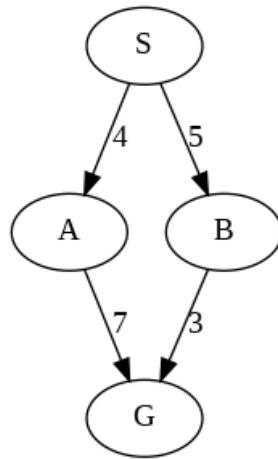


Figure 11.2: Exemplary Acyclic Graph.

```

1 from collections import defaultdict
2 al = defaultdict(list)
3 al['S'] = [( 'A' , 4) , ( 'B' , 5)]
4 al['A'] = [( 'G' , 7)]
5 al['B'] = [( 'G' , 3)]
  
```

With uninformed search, we only know the goal test and the adjacent nodes, but without knowing which non-goal states are better. Assuming and limiting the state space to be a tree for now so that we won't worry about repeated states.

There are generally two ways to order nodes in the frontier without domain-specific information:

- Queue that nodes are first in and first out (FIFO) from the frontier set. This is called breath-first search.
- Stack that nodes are last in but first out (LIFO) from the frontier set. This is called depth-first search.
- Priority queue that nodes are sorted increasingly in the path cost from source to each node from the frontier set. This is called Uniform-Cost Search.

11.2.1 Breath-first Search

Breath-first search always expand the shallowest node in the frontier first, visiting nodes in the tree level by level as illustrated in Fig. 11.3. Using Q to denote the frontier set, the search process is explained:

```

Q=[A]
Expand A, add B and C into Q
  
```

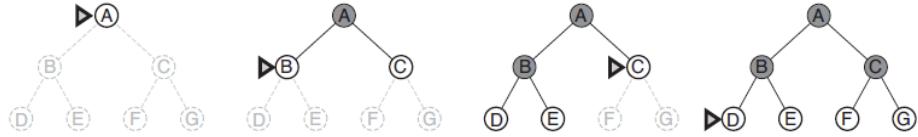


Figure 11.3: Breath-first search on a simple search tree. At each stage, the node to be expanded next is indicated by a marker.

```

Q=[B, C]
Expand B, add D and E into Q
Q=[C, D, E]
Expand C, add F and G into Q
Q=[D, E, F, G]
Finish expanding D
Q=[E, F, G]
Finish expanding E
Q=[F, G]
Finish expanding F
Q=[G]
Finish expanding G
Q=[]

```

The implementation can be done with a FIFO queue iteratively as:

```

1 def bfs(g, s):
2     q = [s]
3     while q:
4         n = q.pop(0)
5         print(n, end = ' ')
6         for v, _ in g[n]:
7             q.append(v)

```

Call the function with parameters as `bfs(al, 'S')`, the output is as:

```
S A B G G
```

Properties Breath-first search is **complete** because it can always find the goal node if it exists in the graph. It is also **optimal** given that all actions(arcs) have the same constant cost, or costs are positive and non-decreasing with depth.

Time Complexity We can clearly see that BFS scans each node in the tree exactly once. If our tree has n nodes, it makes the time complexity $O(n)$. However, the search process can be terminated once the goal is found, which can be less than n . Thus we measure the time complexity by counting the number of nodes expanded while searching is running. Assume the tree has a branching factor b at each non-leaf node and the goal node locates at depth d , we sum up the number of nodes from depth 0 to depth d , the total

number of nodes expanded are:

$$n = \sum_{i=0}^d b^i \quad (11.1)$$

$$= \frac{b^{d+1} - 1}{b - 1} \quad (11.2)$$

Therefore, we have a time complexity of $O(b^d)$. It is usually very slow to find solutions with a large number of steps because it must look at all shorter length possibilities first.

Space Complexity The space is measured in terms of the maximum size of frontier set during the search. In BFS, the maximum size is the number of nodes at depth d , resulting the total space cost to $O(b^d)$.

11.2.2 Depth-first Search

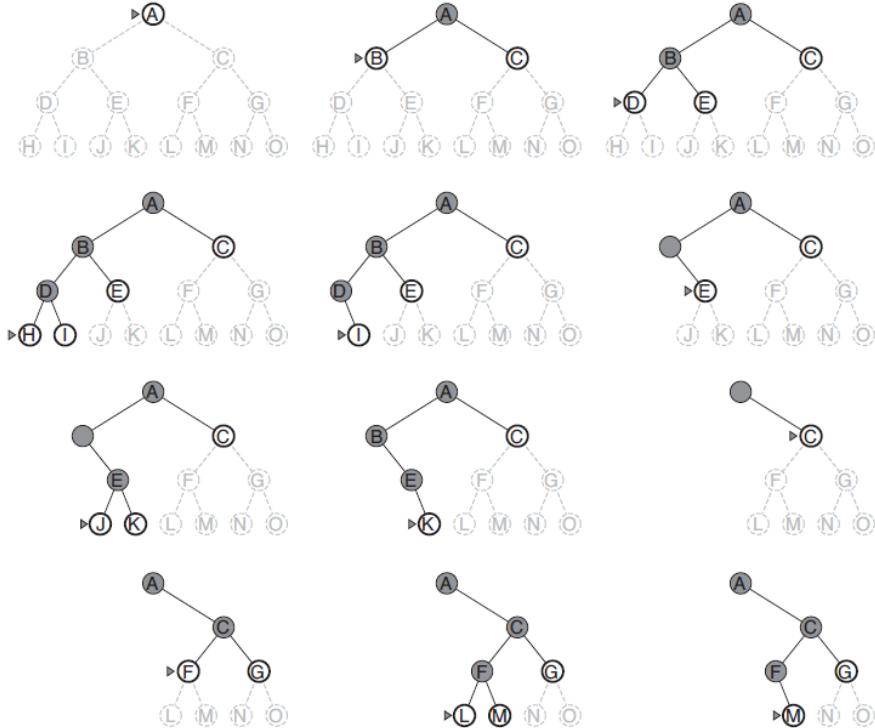


Figure 11.4: Depth-first search on a simple search tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory as node L disappears. Dark gray marks nodes that is being explored but not finished.

Depth-first search on the other hand always expand the deepest node from the frontier first. As shown in Fig. 11.4, Depth-first search starts at the root node and continues branching down a particular path. Using S to denote the frontier set which is indeed a stack, the search process is explained:

```
S=[A]
Expand A, add C and B into S
S=[C, B]
Expand B, add E and D into S
S=[C, E, D]
Expand D
S=[C, E]
Expand E
S=[C]
Expand C, add G and F into S
S=[C, G, F]
Expand F
S=[C, G]
Expand G
S=[C]
Expand C
S=[]
```

Depth-first can be implemented either recursively or iteratively.

Recursive Implementation In the recursive version, the recursive function keeps calling the recursive function itself to expand its adjacent nodes. Starting from a source node, it always deepen down the path until a leaf node is met and then it backtrack to expand its other siblings (or say other adjacent nodes). The code is as:

```
1 def dfs(g, vi):
2     print(vi, end=' ')
3     for v, _ in g[vi]:
4         dfs(g, v)
```

Call the function with parameters as `dfs(al, 'S')`, the output is as:

```
S A G B G
```

Iterative Implementation According to the definition, we can implement DFS with LIFO stack data structure. The code is similar to that of BFS other than using different data structure from the frontier set.

```
1 def dfs_iter(g, s):
2     stack = [s]
3     while stack:
4         n = stack.pop()
5         print(n, end=' ')
6         for v, _ in g[n]:
7             stack.append(v)
```

Call the function with parameters as `dfs_iter(al, 'S')`, the output is as:

```
S B G A G
```

We observe that the ordering is not exactly the same as of the recursive counterpart. To keep the ordering consistent, we simply need to add the adjacent nodes in reversed order. In practice, we replace $g[n]$ with $g[n][:-1]$.

Properties DFS may not terminate without a fixed depth bound to limit the amount of nodes that it expands. DFS is **not complete** because it always deepens the search and in some cases the supply of nodes even within the cutting off fixed depth bound can be infinitely. DFS is **not optimal**, in our example, of our goal node is C, it goes through nodes A, B, D, E before it finds node C. While, in the BFS, it only goes through nodes A and C. However, when we are lucky, DFS can find long solutions quickly.

Time Complexity For DFS, it might need to explore all nodes within graph to find the target, thus its worst-case time and space complexity is not decided upon by the depth of the goal, but the total depth of the graph, d instead. DFS has the same time complexity as BFS, which is $O(b^d)$.

Space Complexity The stack will at most stores a single path from the root to a leaf node (goal node) along with the remaining unexpanded siblings so that when it has visited all children, it can backtrack to a parent node, and know which sibling to explore next. Therefore, the space that needed for DFS is $O(bd)$. In most cases, the branching factor is a constant, which makes the space complexity be mainly influenced by the depth of the search tree. Obviously, DFS has great efficiency in space, which is why it is adopted as the basic technique in many areas of computer science, such as solving constraint satisfaction problems(CSPs). The backtracking technique we are about to introduce even further optimizes the space complexity on the basis of DFS.

11.2.3 Uniform-Cost Search(UCS)

When a priority queue is used to order nodes measured by the path cost of each node to the root in the frontier, this is called uniform-cost search, aka Cheapest First Search. In UCS, frontier set is expanded only in the direction which requires the minimum cost to travel to from root node. UCS only terminates when a path has explored the goal node, and this path is the cheapest path among all paths that can reach to the goal node from the initial point. When UCS is applied to find shortest path in a graph, it is called Dijkstra's Algorithm.

We demonstrate the process of UCS with the example shown in Fig. 11.2.

Here, our source is ‘S’, and the goal is ‘G’. We are set to find a path from source to goal with minimum cost. The process is shown as:

```

Q = [(0, S)]
Expand S, add A and B
Q = [(4, A), (5, B)]
Expand A, add G
Q = [(5, B), (11, G)]
Expand B, add G
Q = [(8, G), (11, G)]
Expand G, goal found, terminate.

```

And the Python source code is:

```

1 import heapq
2 def ucs(graph, s, t):
3     q = [(0, s)] # initial path with cost 0
4     while q:
5         cost, n = heapq.heappop(q)
6         # Test goal
7         if n == t:
8             return cost
9         else:
10            for v, c in graph[n]:
11                heapq.heappush(q, (c + cost, v))
12    return None

```

Properties Uniformed-Cost Search is **complete** as a similar search strategy compared with breath-first search(using queue). It is optimal even if there exist negative edges.

Time and Space Complexity Similar to BFS, both the worst case time and space complexity is $O(b^d)$. When all edge costs are c , and C^* is the best goal path cost, the time and space complexity can be more precisely represented as $O(b^{C^*/c})$.

11.2.4 Iterative-Deepening Search

Iterative-Deepening Search(IDS) is a modification on top of DFS, more specifically depth limited DFS(DLS); as the name suggests, IDS sets a maximum depth as a “depth bound”, and it calls DLS as a subroutine looping from depth zero to maximum depth to expand nodes just as DFS will do and it only does goal test for nodes at the testing depth.

Using the graph in Fig. 11.2 as an example. The process is shown as:

```

maxDepth = 3

depth = 0: S = [S]
Test S, goal not found

depth = 1: S =[S]

```

```

Expand S, S = [B, A]
Test A, goal not found
Test B, goal not found

depth = 2: S=[S]
Expand S, S=[B, A]
Expand A, S=[B, G]
Test G, goal found, STOP

```

The implementation of the DLS goes easier with recursive DFS, we use a count down to variable `maxDepth` in the function, and will only do goal testing until this variable reaches to zero. The code is as:

```

1 def dls(graph, cur, t, maxDepth):
2     # End Condition
3     if maxDepth == 0:
4         if cur == t:
5             return True
6     if maxDepth < 0:
7         return False
8
9     # Recur for adjacent vertices
10    for n, _ in graph[cur]:
11        if dls(graph, n, t, maxDepth - 1):
12            return True
13    return False

```

With the help of function `dls`, the implementation of DLS is just an iterative call to the subroutine:

```

1 def ids(graph, s, t, maxDepth):
2     for i in range(maxDepth):
3         if dls(graph, s, t, i):
4             return True
5     return False

```

Analysis It appears to us that we are undermining the efficiency of the original DFS since the algorithm ends up visiting top level nodes of the goal multiple times. However, it is not as expensive as it seems to be, since in a tree most of the nodes are in the bottom levels. If the goal node locates at the bottom level, DLS will not have an obvious efficiency decline. But if the goal locates on top levels on the right side of the tree, it avoids to visit all nodes across all depths on the left half first and then be able to find this goal node.

Properties Through the depth limited DFS, IDS has advantages of DFS:

- Limited space linear to the depth and branching factor, giving $O(bd)$ as space complexity.
- In practice, even with redundant effort, it still finds longer path more quickly than BFS does.

By iterating through from lower to higher depth, IDS has advantages of BFS, which comes with **completeness** and **optimality** stated the same as of BFS.

Time and Space Complexity The space complexity is the same as of BFS, $O(bd)$. The time complexity is slightly worse than BFS or DFS due to the repetitive visiting nodes on top of the search tree but it still has the same worst case exponential time complexity, $O(b^d)$.

11.2.5 Bidirectional Search**

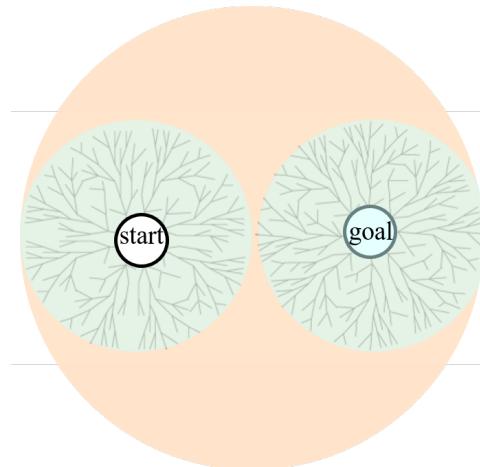


Figure 11.5: Bidirectional search.

Bidirectional search applies breadth-first search from both the start and the goal node, with one BFS from start moving forward and one BFS from the goal moving backward until their frontiers meet. This process is shown in Fig. 11.5. As we see, each BFS process only visit $O(b^{d/2})$ nodes comparing with one single BFS that visits $O(b^d)$ nodes. This will improve both the time and space efficiency by $b^{d/2}$ times compared with vanilla BFS.

Implementation Because the BFS that starts from the goal needs to move backwards, the easy way to do this is to create another copy of the graph wherein each edge has opposite direction compared with the original. By creating a reversed graph, we can use a forward BFS from the goal.

We apply level by level BFS instead of updating the queue one node by one node. For better efficiency of the intersection of the frontier set from both BFS, we use **set** data structure instead of simply a **list** or a FIFO queue.

Use Fig. 11.2 as an example, if our source and goal is ‘S’ and ‘G’ respectively, if we proceed both BFS simultaneously, the process looks like this:

```
qs = ['S']
qt = ['G']
Check intersection , and proceed
qs = ['A', 'B']
qt = ['A', 'B']
Check intersection , frontier meet , STOP
```

No process in this case, however, the above process will end up missing the goal node if we change our goal to be ‘A’. This process looks like:

```
qs = ['S']
qt = ['A']
Check intersection , and proceed
qs = ['A', 'B']
qt = ['S']
Check intersection , and proceed
qs = ['G']
qt = []
STOP
```

This because for source and goal nodes that has a shortest path with even length, if we proceed the search process simultaneously, we will always end up missing the intersection. Therefore, we process each BFS iteratively—one at a time to avoid such troubles.

The code for one level at a time BFS with `set` and for the intersection check is as:

```
1 def bfs_level(graph, q, bStep):
2     if not bStep:
3         return q
4     nq = set()
5     for n in q:
6         for v, c in graph[n]:
7             nq.add(v)
8     return nq
9
10 def intersect(qs, qt):
11     if qs & qt: # intersection
12         return True
13     return False
```

The main code for bidirectional search is as:

```
1 def bis(graph, s, t):
2     # First build a graph with opposite edges
3     bgraph = defaultdict(list)
4     for key, value in graph.items():
5         for n, c in value:
6             bgraph[n].append((key, c))
7     # Start bidirectional search
8     qs = {s}
```

```

9   qt = {t}
10  step = 0
11  while qs and qt:
12      if intersect(qs, qt):
13          return True
14      qs = bfs_level(graph, qs, step%2 == 0)
15      qt = bfs_level(bgraph, qt, step%2 == 1)
16      step = 1 - step
17  return False

```

11.2.6 Summary

Table 11.1: Performance of Search Algorithms on Trees or Acyclic Graph

Method	Complete	Optimal	Time	Space
BFS	Y	Y, if	$O(b^d)$	$O(b^d)$
UCS	Y	Y	$O(C^*/c)$	$O(C^*/c)$
DFS	N	N	$O(b^m)$	$O(bm)$
IDS	Y	Y, if	$O(b^d)$	$O(bd)$
Bidirectional Search	Y	Y, if	$O(b^{d/2})$	$O(b^{d/2})$

Using b as branching factor, d as the depth of the goal node, and m is the maximum graph depth. The properties and complexity for the five uninformed search strategies are summarized in Table. 11.1.

11.3 Graph Search

Cycles This section is devoted to discuss more details about two search strategies—BFS and DFS in more general graph setting. In the last section, we just assumed our graph is either a tree or acyclic directional graph. In more general real-world setting, there can be cycles within a graph which will lead to infinite loops of our program.

Print Paths Second, we talked about the paths, but we never discuss how to track all the paths. In this section, we would like to see how we can track paths first, and then with the tracked paths, we detect cycles to avoid getting into infinite loops.

More Efficient Graph Search Third, the last section is all about tree search, however, in a large graph, this is not efficient by visiting some nodes multiple times if they happen to be on the multiple paths between the source and any other node in the graph. Usually, depends on the application scenarios, graph search which remembers already-expanded nodes/states in the graph and avoids expanding again by checking any about to be expanded

node to see if it exists in frontier set or the explored set. This section, we introduce graph search that suits for general purposed graph problems.

Visiting States We have already explained that we can use three colors: WHITE, GREY, and BLACK to denote nodes within the unexpanded, frontier, and explored set, respectively. We are doing so to avoid the hassles of tracking three different sets, with visiting state, it is all simplified to a color check. We define a STATE class for convenience.

```
class STATE:
    white = 0
    gray = 1
    black = 2
```

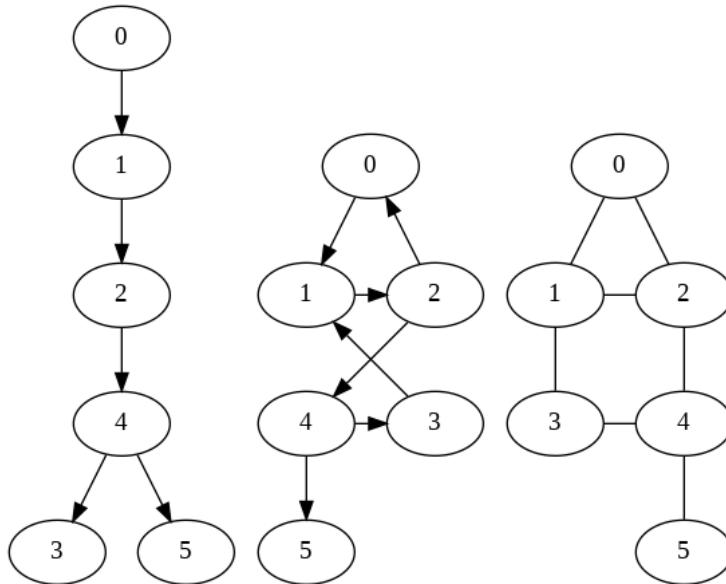


Figure 11.6: Exemplary Graph: Free Tree, Directed Cyclic Graph, and Undirected Cyclic Graph.

In this section, we use Fig. 11.6 as our exemplary graphs. Each's data structure is defined as:

- Free Tree:

```
1 ft = [[1], [2], [4], [], [3, 5], []]
```

- Directed Cyclic Graph:

```
1 dcg = [[1], [2], [0, 4], [1], [3, 5], []]
```

- Undirected Cyclic Graph

```

1 ucg = [[1, 2], [0, 2, 3], [0, 1, 4], [1, 4], [2, 3, 5],
        [4]]

```

Search Tree It is important to realize the Searching ordering is always forming a tree, this is terminologized as **Search Tree**. In a tree structure, the search tree is itself. In a graph, we need to figure out the search tree and it decides our time and space complexity.

11.3.1 Depth-first Search in Graph

In this section we will further the depth-first tree search and explore depth-first graph search to compare their properties and complexity.

Depth-first Tree Search

Vanilla Depth-first Tree Search Our previous code slightly modified to suit for the new graph data structure works fine with the free tree in Fig. 11.6. The code is as:

```

1 def dfs(g, vi):
2     print(vi, end=' ')
3     for nv in g[vi]:
4         dfs(g, nv)

```

However, if we call it on the cyclic graph, `dfs(dcg, 0)`, it runs into stack overflow.

Cycle Avoiding Depth-first Tree Search So, how to avoid cycles? We know the definition of a cycle is a closed path that has at least one node that repeats itself; in our failed run, we were stuck with cycle [0, 1, 2, 0]. Therefore, let us add a `path` in the recursive function, and whenever we want to expand a node, we check if it forms a cycle or not by checking the membership of a candidate to nodes comprising the path. We save all paths and the visiting ordering of nodes in two lists: `paths` and `orders`. The recursive version of code is:

```

1 def dfs(g, vi, path):
2     paths.append(path)
3     orders.append(vi)
4     for nv in g[vi]:
5         if nv not in path:
6             dfs(g, nv, path+[nv])
7     return

```

Now we call function `dfs` for `ft`, `dcg`, and `ucg`, the `paths` and `orders` for each example is listed:

- For the free tree and the directed cyclic graph, they have the same output. The `orders` are:

```
[0, 1, 2, 4, 3, 5]
```

And the paths are:

```
[[0], [0, 1], [0, 1, 2], [0, 1, 2, 4], [0, 1, 2, 4, 3], [0, 1, 2, 4, 5]]
```

- For the undirected cyclic graph, orders are:

```
[0, 1, 2, 4, 3, 5, 3, 4, 2, 5, 2, 1, 3, 4, 5, 4, 3, 1, 5]
```

And the paths are:

```
[[0],
[0, 1],
[0, 1, 2],
[0, 1, 2, 4],
[0, 1, 2, 4, 3],
[0, 1, 2, 4, 5],
[0, 1, 3],
[0, 1, 3, 4],
[0, 1, 3, 4, 2],
[0, 1, 3, 4, 5],
[0, 2],
[0, 2, 1],
[0, 2, 1, 3],
[0, 2, 1, 3, 4],
[0, 2, 1, 3, 4, 5],
[0, 2, 4],
[0, 2, 4, 3],
[0, 2, 4, 3, 1],
[0, 2, 4, 5]]
```

These paths mark the search tree, we visualize the search tree for each exemplary graph in Fig. 11.7.

Depth-first Graph Search

We see that from the above implementation, for a graph with only 6 nodes, we have been visiting nodes for a total of 19 times. A lot of nodes have been repeating. 1 appears 3 times, 3 appears 4 times, and so on. As we see the visiting order being represented with a **search tree** in Fig. 11.7, our complexity is getting close to $O(b^h)$, where b is the branching factor and h is the total vertices of the graph, marking the upper bound of the maximum depth that the search can traverse. If we simply want to search if a value or a state exists in the graph, this approach insanely complicates the situation. What we do next is to avoid revisiting the same vertex again and again by tracking the visiting state of a node.

In the implementation, we only track the longest path—from source vertex to vertex that has no more unvisited adjacent vertices.

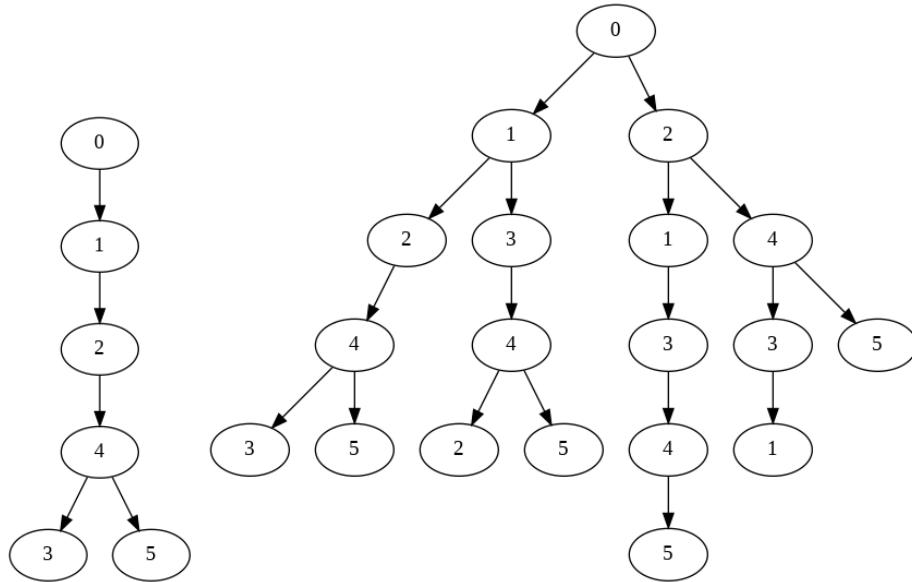


Figure 11.7: Search Tree for Exemplary Graph: Free Tree and Directed Cyclic Graph, and Undirected Cyclic Graph.

```

1 def dfgs(g, vi, visited, path):
2     visited.add(vi)
3     orders.append(vi)
4     bEnd = True # node without unvisited adjacent nodes
5     for nv in g[vi]:
6         if nv not in visited:
7             if bEnd:
8                 bEnd = False
9                 dfgs(g, nv, visited, path + [nv])
10            if bEnd:
11                paths.append(path)

```

Now, we call this function with ucg as:

```

1 paths, orders = [], []
2 dfgs(ucg, 0, set(), [0])

```

The output for `paths` and `orders` are:

```
([[0, 1, 2, 4, 3], [0, 1, 2, 4, 5]], [0, 1, 2, 4, 3, 5])
```

Did you notice that the depth-first graph search on the undirected cyclic graph shown in Fig. 11.6 has the same visiting order of nodes and same search tree as the free tree and directed cyclic graph in Fig. 11.6?

Efficient Path Backtrace In graph search, each node is added into the frontier and expanded only once, and the search tree of a $|V|$ graph will only have $|V| - 1$ edges. Tracing paths by saving each path as a list in the frontier set is costly; for a partial path in the search tree, it is repeating itself

multiple times if it happens to be part of multiple paths, such as partial path $0 \rightarrow 1 \rightarrow 2 \rightarrow 4$. We can bring down the memory cost to $O(|v|)$ if we only save edges by using a `parent` dict with key and value referring as the node and its parent node in the path, respectively. For example, edge $0 \rightarrow 1$ is saved as `parent[1] = 0`. Once we find out goal state, we can backtrace from this goal state to get the path. The backtrace code is:

```

1 def backtrace(s, t, parent):
2     p = t
3     path = []
4     while p != s:
5         path.append(p)
6         p = parent[p]
7     path.append(s)
8     return path[::-1]
```

Now, we modify the dfs code as follows to find a given state (vertex) and obtaining the path from source to target:

```

1 def dfgs(g, vi, s, t, visited, parent):
2     visited.add(vi)
3     if vi == t:
4         return backtrace(parent, s, t)
5
6     for nv in g[vi]:
7         if nv not in visited:
8             parent[nv] = vi
9             fpath = dfgs(g, nv, s, t, visited, parent)
10            if fpath:
11                return fpath
12
13 return None
```

The whole Depth-first graph search tree constructed from the `parent` dict is delineated in Fig. 11.8 on the given example.

Properties The completeness of DFS depends on the search space. If your search space is finite, then Depth-First Search is complete. However, if there are infinitely many alternatives, it might not find a solution. For example, suppose you were coding a path-search problem on city streets, and every time your partial path came to an intersection, you always searched the left-most street first. Then you might just keep going around the same block indefinitely.

The depth-first graph search is **nonoptimal** just as Depth-first tree search. For example, if the task is to find the shortest path from source 0 to target 2. The shortest path should be $0 \rightarrow 2$, however depth-first graph search will return $0 \rightarrow 1 \rightarrow 2$. For the search tree using depth-first tree search, it can find the shortest path from source 0 to 2. However, it will explore the whole left branch starts from 1 before it finds its goal node on the right side.

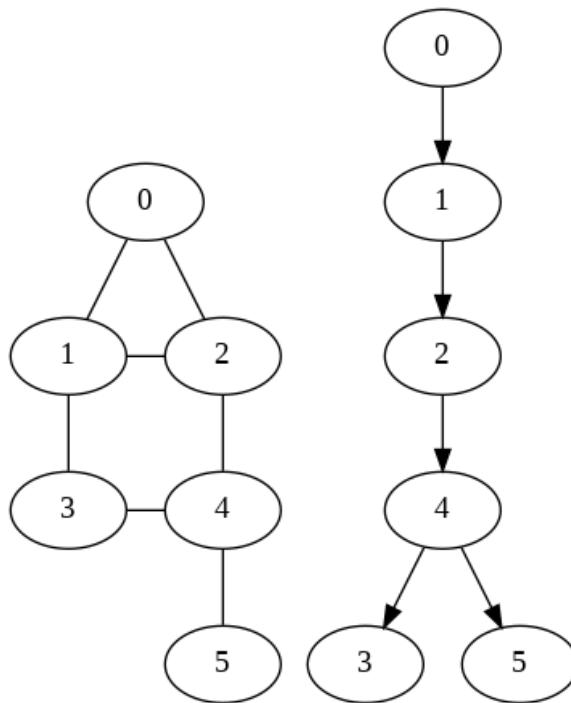


Figure 11.8: Depth-first Graph Search Tree.

Time and Space Complexity For the depth-first graph search, we use aggregate analysis. The search process covers all edges, $|E|$ and vertices, $|V|$, which makes the time complexity as $O(|V| + |E|)$. For the space, it uses space $O(|V|)$ in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices.

Applications

Depth-first tree search is adopted as the basic workhorse of many areas of AI, such as solving CSP, as it is a brute-force solution. In Chapter Combinatorial Search, we will learn how “backtracking” technique along with others can be applied to speed things up. Depth-first graph search is widely used to solve graph related tasks in non-exponential time, such as Cycle Check(linear time) and shortest path.



Questions to ponder:

- Only track the longest paths.
- How to trace the edges of the search tree?
- Implement the iterative version of the recursive code.

11.3.2 Breath-first Search in Graph

We further breath-first tree search and explore breath-first graph search in this section to grasp better understanding of one of the most general search strategies. Because that BFS is implemented iteratively, the implementation in this section of sheds light to the iterative counterparts of DFS's recursive implementations from last section.

Breath-first Tree Search

Similarly, out vanilla breath-first tree search shown in Section. ?? will get stuck with the cyclic graph in Fig. 11.6.

Cycle Avoiding Breath-first Tree Search We avoid cycles with similar strategy to DFS tree search that traces paths and checks membership of node. In BFS, we track paths by explicitly adding paths to the `queue`. Each time we expand from the frontier (queue), the node we need is the last item in the path from the queue. In the implementation, we only track the longest paths from the search tree and the visiting orders of nodes. The Python code is:

```

1 def bfs(g, s):
2     q = [[s]]
3     paths, orders = [], []
4     while q:
5         path = q.pop(0)
6         n = path[-1]
7         orders.append(n)
8         bEnd = True
9         for v in g[n]:
10             if v not in path:
11                 if bEnd:
12                     bEnd = False
13                     q.append(path + [v])
14             if bEnd:
15                 paths.append(path)
16     return paths, orders

```

Now we call function `bfs` for `ft`, `dcg`, and `ucg`, the `paths` and `orders` for each example is listed:

- For the free tree and the directed cyclic graph, they have the same output. The `orders` are:

```
[0, 1, 2, 4, 3, 5]
```

And the `paths` are:

```
[[0, 1, 2, 4, 3], [0, 1, 2, 4, 5]]
```

- For the undirected cyclic graph, `orders` are:

```
[0, 1, 2, 2, 3, 1, 4, 4, 3, 3, 5, 3, 5, 2, 5, 4, 1, 5]
```

And the paths are:

```
[[0, 2, 4, 5], [0, 1, 2, 4, 3], [0, 1, 2, 4, 5], [0, 1, 3, 4, 2], [0, 1, 3, 4, 5], [0, 2, 4, 3, 1], [0, 2, 1, 3, 4, 5]]
```

Properties We can see the visiting orders of nodes are different from Depth-first tree search counterparts. However, the corresponding search tree for each graph in Fig. 11.6 is the same as its counterpart—Depth-first Tree Search illustrated in Fig. 11.7. This highlights how different searching strategies differ by visiting ordering of nodes but not differ at the search-tree which depicts the search space—all possible paths.

Applications However, the Breath-first Tree Search and path tracing is extremely more costly compared with DFS counterpart. When our goal is to enumerate paths, go for the DFS. When we are trying to find shortest-paths, mostly use BFS.

Breath-first Graph Search

Similar to Depth-first Graph Search, we use a `visited` set to make sure each node is only added to the frontier(queue) once and thus expanded only once.

BFGS Implementation The implementation of Breath-first Graph Search with goal test is:

```
1 def bfgs(g, s, t):
2     q = [s]
3     parent = {}
4     visited = {s}
5     while q:
6         n = q.pop(0)
7         if n == t:
8             return backtrace(s, t, parent)
9         for v in g[n]:
10            if v not in visited:
11                q.append(v)
12                visited.add(v)
13                parent[v] = n
14    return parent
```

Now, use the undirected cyclic graph as example to find the path from source 0 to target 5:

```
1 bfgs(ucg, 0, 5)
```

With the found path as:

```
[0, 2, 4, 5]
```

While this found path is the shortest path between the two vertices measured by the length. The whole Breath-first graph search tree constructed from the `parent` dict is delineated in Fig. 11.9 on the given example.

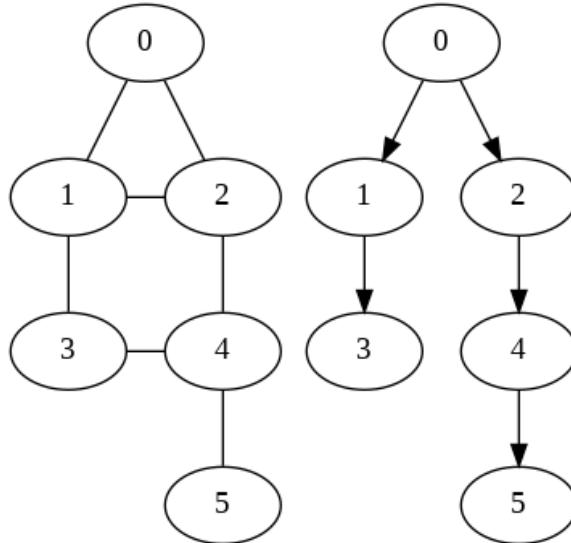


Figure 11.9: Breath-first Graph Search Tree.

Time and Space Complexity Same to DFGS, the time complexity as $O(|V| + |E|)$. For the space, it uses space $O(|V|)$ in the worst case to store vertices on the current search path, the set of already-visited vertices, as well as the dictionary used to store edge relations. The shortage that comes with costly memory usage of Breath-first Graph Search to Depth-first Graph Search is less obvious compared to Breath-first Tree Search to Depth-first Graph Search.

Tree Search VS Graph Search

There are two important characteristics about tree search and graph search:

- Within a graph $G = (V, E)$, either it is undirected or directed, acyclic or cyclic, both the breath-first and depth-first tree search results the same search tree: They both enumerate all possible states (paths) of the search space.
- The conclusion is different for breath-first and depth-first graph search. For acyclic and directed graph (tree), both search strategies result the

same search tree. However, whenever there exists cycles, the depth-first graph search tree might differ from the breath-first graph search tree.

11.3.3 Depth-first Graph Search

Within this section and the next, we focus on explaining more characteristics of the graph search that avoids repeatedly visiting a vertex. Seemingly these features and details are not that useful judging from current context, but we will see how it can be applied to solve problems more efficiently in Chapter Advanced Graph Algorithms, such as detecting cycles, topological sort, and so on.

As shown in Fig. 11.10 (a directed graph), we start from 0, mark it gray, and visit its first unvisited neighbor 1, mark 1 as gray, and visit 1's first unvisited neighbor 2, then 2's unvisited neighbor 4, 4's unvisited neighbor 3. For node 3, it doesn't have white neighbors, we mark it to be complete with black. Now, here, we "backtrack" to its predecessor, which is 4. And then we keep the process till 5 become gray. Because 5 has no edge out any more, it becomes black. Then the search backtracks to 4, to 2, to 1, and eventually back to 0. We should notice the ordering of vertices become gray or black is different. From the figure, the gray ordering is [0, 1, 2, 4, 3, 5], and for the black is [3, 5, 4, 2, 1, 0]. Therefore, it is necessary to distinguish the three states in the depth-first graph search at least.

Three States Recursive Implementation We add additional `colors` list to track the color of each vertices, `orders` to track the ordering of the gray, and `complete_orders` for ordering vertices by their ordering of turning into black—when all of a node's neighbors become black which is after the recursive call in the code.

```

1 def dfs(g, s, colors, orders, complete_orders):
2     colors[s] = STATE.gray
3     orders.append(s)
4     for v in g[s]:
5         if colors[v] == STATE.white:
6             dfs(g, v, colors, orders, complete_orders)
7     colors[s] = STATE.black
8     complete_orders.append(s)
9     return

```

Now, we try to call the function with the undirected cyclic graph in Fig. 11.6.

```

1 v = len(ucg)
2 orders, complete_orders = [], []
3 colors = [STATE.white] * v
4 dfs(ucg, 0, colors, orders, complete_orders)

```

Now, the `orders` and `complete_orders` will end up differently:

```
[0, 1, 2, 4, 3, 5] [3, 5, 4, 2, 1, 0]
```

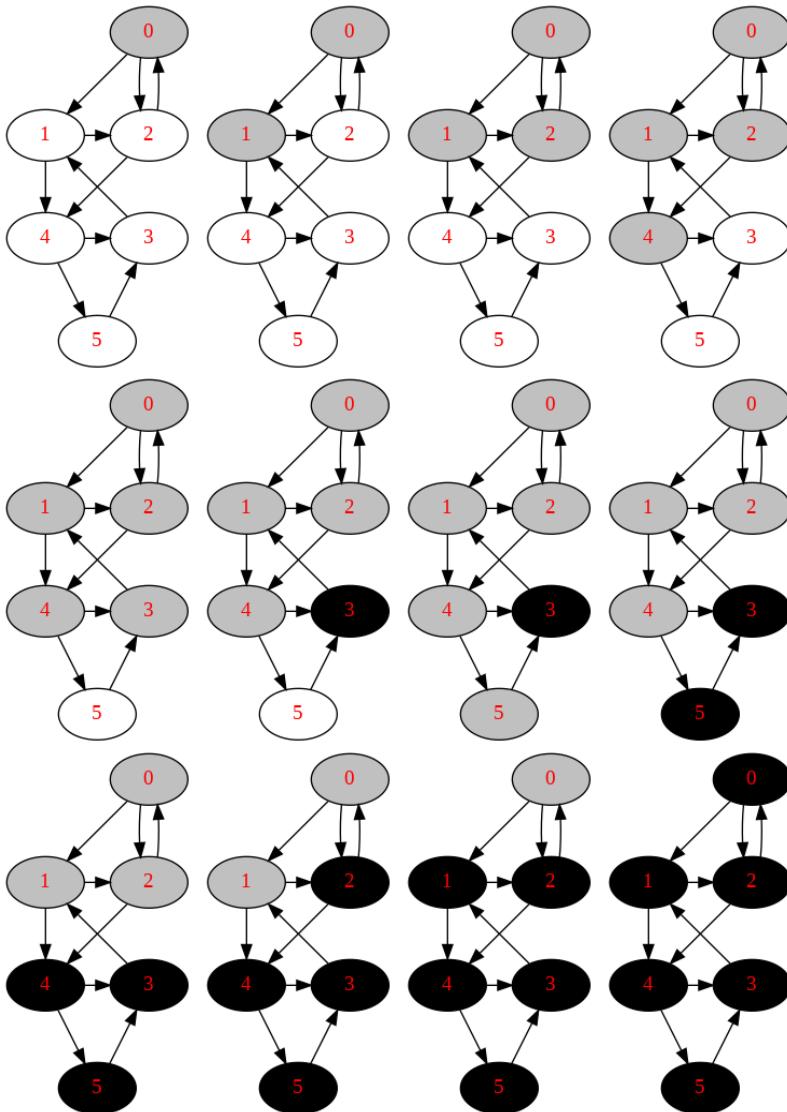


Figure 11.10: The process of Depth-first Graph Search in Directed Graph. The black arrows denotes the the relation of u and its not visited neighbors v . And the red arrow marks the backtrack edge.

Three States and Edges Depth-first Graph Search on graph $G = (V, E)$ connects all reachable vertices from a given source in the graph in the form of a depth-first forest G_π . Edges within G_π are called **tree edges**. Tree edges are edges marked with black arrows in Fig. 11.11. Other edges in G can be classified into three categories based on Depth-first forest G_π , they are:

1. Back edges which connect a node back to one of its ancestors in the

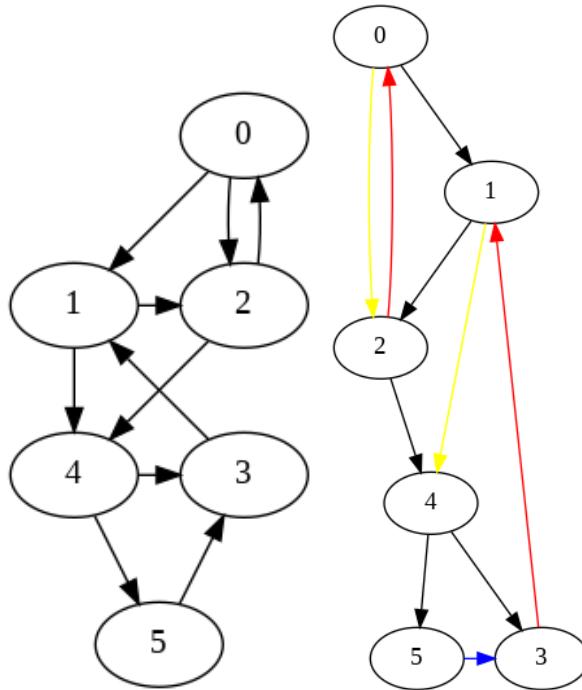


Figure 11.11: Classification of Edges: black marks tree edge, red marks back edge, yellow marks forward edge, and blue marks cross edge.

depth-first forest G_π . Marked as red edges in Fig. 11.11.

2. Forward edges point from a node to one of its descendants in the depth-first forest G_π . Marked as yellow edges in Fig. 11.11.
3. Cross edges point from a node to a previously visited node that is neither an ancestor nor a descendant in the depth-first forest G_π . Marked as blue edges in Fig. 11.11.

We can decide the type of tree edge using the DFS execution with the states: for an edge (u, v) , depends on whether we have visited v before in the DFS and if so, the relationship between u and v .

1. If v is WHITE, then the edge is a tree edge.
2. If v is GRAY—both u and v are both being visited—then the edge is a back edge. In directed graph, this indicates that we meet a cycle.
3. If v is BLACK, that v is finished, and that the $\text{start_time}[u] < \text{start_time}[v]$, then the edge is a forward edge.
4. If v is BLACK, but the $\text{start_time}[u] > \text{start_time}[v]$, then the edge is a cross edge.

In undirected graph, there is no forward edge or cross edge. Therefore, it does not really need three colors. Usually, we can simply mark it as visited or not visted.

Classification of edges provide important information about the graph, e.g. to if we detect a back edge in directed graph, we find a cycle.

Parenthesis Structure In either undirected or directed graph, the discovered time when state goes from WHITE to GRAY and the finish time when state turns to BLACK from GRAY has the parenthesis structure. We modify `dfs` to track the time: a static variable `t` is used to track the time, `discover` and `finish` is used to record the first discovered and finished time. The implementation is shown:

```

1 def dfs(g, s, colors):
2     dfs.t += 1 # static variable
3     colors[s] = STATE.gray
4     dfs.discover[s] = dfs.t
5     for v in g[s]:
6         if colors[v] == STATE.white:
7             dfs(g, v, colors)
8     # complete
9     dfs.t += 1
10    dfs.finish[s] = dfs.t
11

```

Now, we call the above function with directed graph in Fig. 11.11.

```

1 v = len(dcg)
2 colors = [STATE.white] * v
3 dfs.t = -1
4 dfs.discover, dfs.finish = [-1] * v, [-1] * v
5 dfs(dcg, 0, colors)

```

The output for `dfs.discover` and `dfs.finish` are:

```
([0, 1, 2, 4, 3, 6], [11, 10, 9, 5, 8, 7])
```

From `dfs.discover` and `dfs.finish` list, we can generate a new list of merged order, `merge_orders` that arranges nodes in order of there discovered and finish time. The code is as:

```

1 def parenthesis(dt, ft, n):
2     merge_orders = [-1] * 2 * n
3     for v, t in enumerate(dt):
4         merge_orders[t] = v
5     for v, t in enumerate(ft):
6         merge_orders[t] = v
7
8     print(merge_orders)
9     nodes = set()
10    for i in merge_orders:
11        if i not in nodes:
12            print('(', i, end = ', ')

```

```

13     nodes.add(i)
14 else:
15     print(i, ' ', ', end = ' ')

```

The output is:

```

1 [0, 1, 2, 4, 3, 3, 5, 6, 6, 5, 4, 2, 1, 0]
2 ( 0, ( 1, ( 2, ( 4, ( 3, 3 ), ( 5, ( 6, 6 ), 5 ), 4 ), 2 ), 1 ),
0 ),

```

We would easily find out that the ordering of nodes according to the discovery and finishing time makes a well-defined expression in the sense that the parentheses are properly nested.



Questions to ponder:

- Implement the iterative version of the recursive code.

11.3.4 Breadth-first Graph Search

We have already known how to implement BFS of both the tree and graph search versions. In this section, we want to first exemplify the state change process of BFGS with example shown in Fig. 11.10. Second, we focus on proving that within the breath-first graph search tree, a path between root and any other node is the shortest path.

Three States Iterative Implementation When a node is first put into the frontier set, it is marked with gray color. A node is complete only if all its adjacent nodes turn into gray or black. With the visiting ordering of the breath-first graph search, the state change of nodes in the search process is shown in Fig. 11.12. The Python code is:

```

1 def bfgs_state(g, s):
2     v = len(g)
3     colors = [STATE.white] * v
4
5     q, orders = [s], [s]
6     complete_orders = []
7     colors[s] = STATE.gray # make the state of the visiting node
8     while q:
9         u = q.pop(0)
10        for v in g[u]:
11            if colors[v] == STATE.white:
12                colors[v] = STATE.gray
13                q.append(v)
14                orders.append(v)
15
16        # complete
17        colors[u] = STATE.black
18        complete_orders.append(u)
19    return orders, complete_orders

```

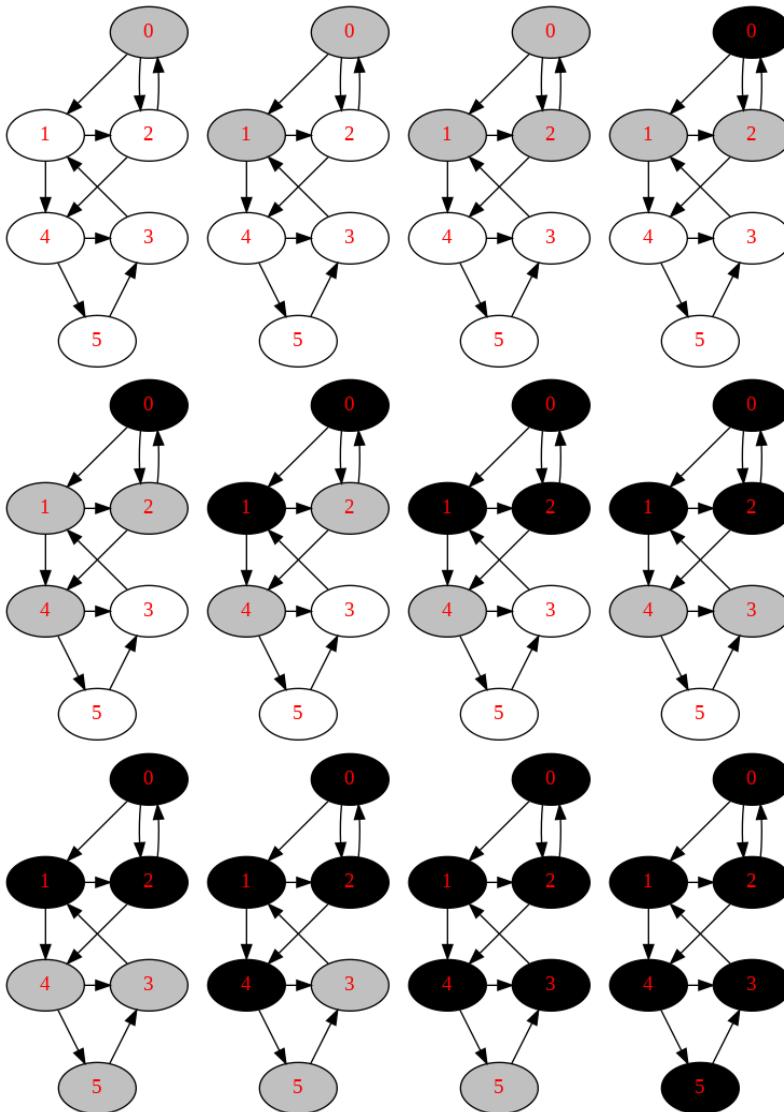


Figure 11.12: The process of Breadth-first Graph Search. The black arrows denotes the the relation of u and its not visited neighbors v . And the red arrow marks the backtrack edge.

The printout of `orders` and `complete_orders` are:

```
([0, 1, 2, 4, 3, 5], [0, 1, 2, 4, 3, 5])
```

Properties In breath-first graph search, the first discovery and finishing time are different for each node, but the discovery ordering and the finishing ordering of nodes are the same ordering.

Shortest Path

Applications

The common problems that can be solved by BFS are those only need one solution: the best one such like getting the shortest path. As we will learn later that breath-first-search is commonly used as archetype to solve graph optimization problems, such as Prim's minimum-spanning-tree algorithm and Dijkstra's single-source-paths algorithm.

11.4 Tree Traversal

11.4.1 Depth-First Tree Traversal

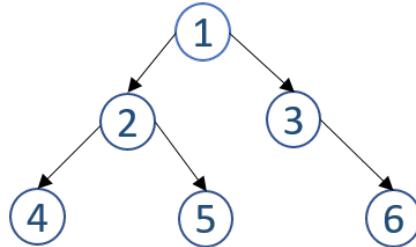


Figure 11.13: Exemplary Binary Tree

Introduction

Depth-first search starts at the root node and continues branching down a particular path; it selects a child node that is at the deepest level of the tree from the frontier to expand next and defers the expansion of this node's siblings. Only when the search hits a dead end (a node that has no child) does the search “backtrack” to its parent node, and continue to branch down to other siblings that were deferred. A recursive tree can be traversed recursively. We print out the value of current node, then apply recursive call on the left and right node; by treating each node as a subtree, naturally a recursive call to a node can be thought of handling the traversal of that subtree. The code is quite straightforward:

```

1 def recursive(node):
2     if not node:
3         return
4     print(node.val, end=' ')
5     recursive(node.left)
6     recursive(node.right)
  
```

Now, we call this function with a tree as shown in Fig. 11.13, the output that indicates the traversal order is:

```
1 1 2 4 5 3 6
```

Three Types of Depth-first Tree Traversal

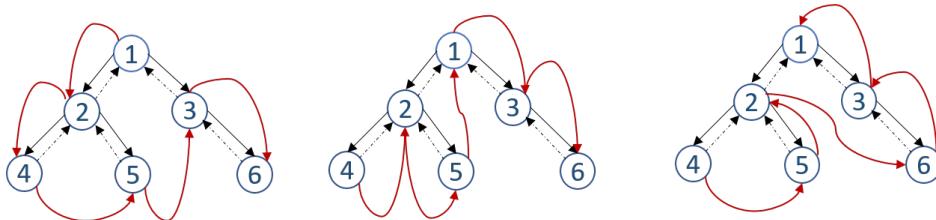


Figure 11.14: Left: PreOrder, Middle: InOrder, Right: PostOrder. The red arrows marks the traversal ordering of nodes.

The visiting ordering between the current node, its left child, and its right child decides the following different types of recursive tree traversals:

- Preorder Traversal with ordering of `[current node, left child, right child]`: it visits the nodes in the tree with ordering [1, 2, 4, 5, 3, 6]. In our example, the recursive function first prints the root node 1, then goes to its left child, which prints out 2. Then it goes to node 4. From node 4, it next moves to its left child which is empty and leads to the termination of the recursive call and then the recursion backward to node 4. Since node 4 has no right child, it further backwards to node 2, and then it checks 2's right child 5. The same process of node 4 happens on node 5. It backwards to node 2, backwards to node 1, and keep visiting its right child 3, and the process goes on. We draw out this process in Fig. 11.14.
- Inorder Traversal with ordering of `[left child, current node, right child]`: it traverses the nodes in ordering of [4, 2, 5, 1, 3, 6]. Three segments will appear with the inorder traversal for a root node: nodes in left subtree, root, and nodes in the right subtree.
- Postorder Traversal with ordering of `[left child, right child, current node]`: it traverses the nodes in ordering of [4, 5, 2, 6, 3, 1].

We offer the code of Inorder Traversal:

```
1 def inorder_traversal(node):
2     if not node:
3         return
4     inorder_traversal(node.left)
5     print(node.val, end=' ')
6     inorder_traversal(node.right)
```



Try to check the other two orderings: [left child, current node, right child] and [left child, right child, current node] by hand first and then write the code to see if you get it right?

Return Values

Here, we want to do the task in a different way: We do not want to just print out the visiting orders, but instead write the ordering in a list and return this list. How would we do it? The process is the same, other than we need to return something(not `None` which is default in Python). If we only have empty node, it shall return us an empty list `[]`, if there is only one node, returns `[1]` instead.

Let us use PreOrder traversal as an example. To make it easier to understand, the same queen this time wants to do the same job in a different way, that she wants to gather all the data from these different states to her own hand. This time, she assumes the two generals A and B will return a list of the subtree, safely and sound. Her job is going to combine the list returned from the left subtree, her data, and the list returned from the right subtree. Therefore, the left general brings back $A = [2, 4, 5]$, and the right general brings back $B = [3, 6]$. Then the final result will be $queue + A + B = [1, 2, 4, 5, 3, 6]$. The Python code is given:

```

1 def PreOrder(root):
2     if root is None:
3         return []
4     ans = []
5     left = PreOrder(root.left)
6     right = PreOrder(root.right)
7     ans = [root.val] + left + right
8     return ans

```

An Example of Divide and Conquer Be able to understand the returned value and combine them is exactly the method of `divide and conquer`, one of the fundamental algorithm design principles. This is a seemingly trivial change, but it approaches the problem solving from a totally different angle: atomic searching to divide and conquer that highlights the structure of the problem. The printing traversal and returning traversal represents two types of problem solving: the first is through searching–searching and treating each node more separately and the second is through reduce and conquer–reducing the problem to a series of smaller subproblems(subtrees where the smallest are empty subtrees) and construct the result by using the information of current problem and the solutions of the subproblems.

Complexity Analysis

It is straightforward to see that it only visit all nodes twice, one in the forward pass and the other in the backward pass of the recursive call, making the time complexity linear to total number of nodes, $O(n)$. The other way is through the recurrence relation, we would write $T(n) = 2 \times T(n/2) + O(1)$, which gives out $O(n)$ too.

11.4.2 Iterative Tree Traversal

In Chapter Iteration and Recursion, we would know that the recursive function might suffer from the stack overflow, and in Python the recursion depth is 1000. This section, we explore iterative tree traversals corresponding to PreOrder, InOrder, and PostOrder tree traversal. We know that the recursion is implemented implicitly with call stack, therefore in our iterative counterparts, they all use an explicit stack data structure to mimic the recursive behavior.

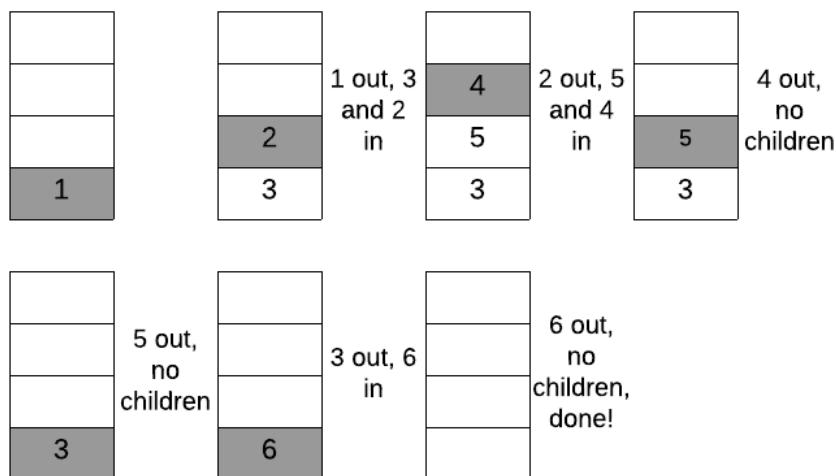


Figure 11.15: The process of iterative preorder tree traversal.

Simple Iterative Preorder Traversal If we know how to implement a DFS iteratively with stack in a graph, we know our iterative preorder traversal. In this version, the stack saves all our frontier nodes.

- At first, we start from the root, and put it into the stack, which is 1 in our example.
- Our frontier set has only one node, thus we have to pop out node 1 and expand the frontier set. When we are expanding node 1, we add its children into the frontier set by pushing them into the stack. In

the preorder traversal, the left child should be first expanded from the frontier stack, indicating we should push the left child into the stack afterward the right child is pushed into. Therefore, we add node 3 and 2 into the stack.

- We continue step 2. Each time, we expand the frontier stack by pushing the toppest node's children into the stack and after popping out this node. This way, we use the first come last ordering of the stack data structure to replace the recursion.

We illustrate this process in Fig. 11.15. The code is shown as:

```

1 def PreOrderIterative(root):
2     if root is None:
3         return []
4     res = []
5     stack = [root]
6     while stack:
7         tmp = stack.pop()
8         res.append(tmp.val)
9         if tmp.right:
10             stack.append(tmp.right)
11         if tmp.left:
12             stack.append(tmp.left)
13

```

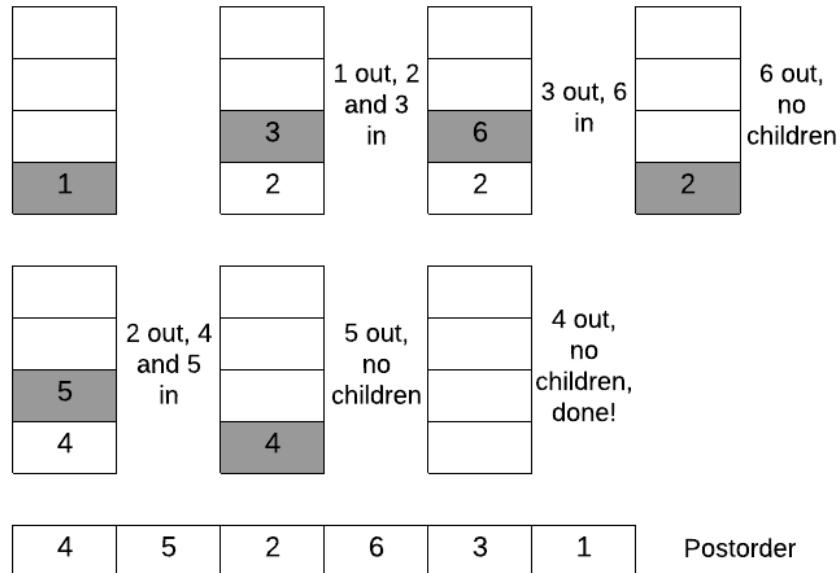


Figure 11.16: The process of iterative postorder tree traversal.

Simple Iterative Postorder Traversal Similar to the above preorder traversal, the postordering is the ordering of nodes finishing the expanding of both its left and right subtree, thus with the ordering of `left subtree`, `right subtree`, and `root`. In preorder traversal, we obtained the ordering of `root`, `left subtree`, and `right subtree`. We try to reverse the ordering, it becomes `right subtree`, `left subtree`, and `root`. This ordering only differs with postorder by a single a swap between the left and right subtree. So, we can use the same process as in the preorder traversal but expanding a node's children in the order of left and right child instead of right and left. And then the reversed ordering of items being popped out is the postoder traversal ordering. The process is shown in Fig. 11.16. The Python implementation is shown as:

```

1 def PostOrderIterative (root):
2     if root is None:
3         return []
4     res = []
5     stack = [root]
6     while stack:
7         tmp = stack.pop()
8         res.append(tmp.val)
9         if tmp.left:
10            stack.append(tmp.left)
11        if tmp.right:
12            stack.append(tmp.right)
13    return res[::-1]
```

General Iterative Preorder and Inorder Traversal In the depth-first-traversal, we always branch down via the left child of the node at the deepest level in the frontier. The branching only stops when it can no longer find a left child for the deepest node in the frontier. Only till then, it will look around at expanding the right child of this deepest node, and if no such right child exists, it backtracks to its parents node and continues to check its right child to continue the branching down process.

Inspired by this process, we use a pointer, say `cur` to point to the root node of the tree, and we prepare an empty `stack`. The iterative process is:

- The branching down process can be implemented with visiting `cur` node, and pushing it into the `stack`. And then we set `cur=cur.left`, so that it keeps deepening down.
- When one branch down process terminates, we pop out a node from `stack`, and we set `cur=node.right`, so that we expand the branching process to its right sibling.

We illustrate this process in Fig. 11.17. The ordering of items pushed into the stack is the preorder traversal ordering, which is [1, 2, 4, 5, 3, 6]. And

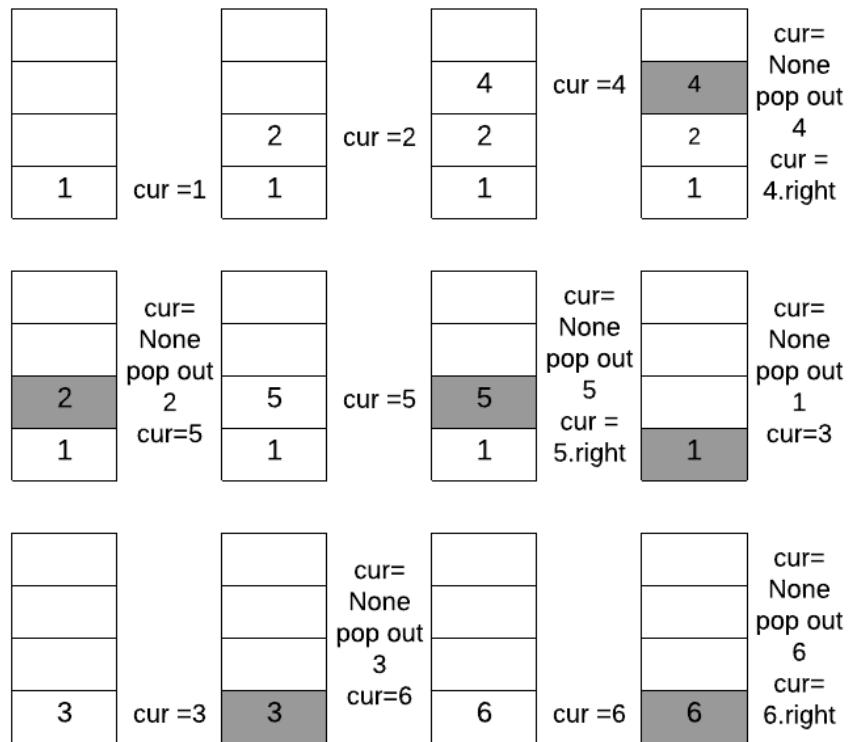


Figure 11.17: The process of iterative tree traversal.

the ordering of items being popped out of the stack is the inorder traversal ordering, which is [4, 2, 5, 1, 3, 6].

Implementation We use two lists—`preorders` and `inorders`—to save the traversal orders. The Python code is:

```

1 def iterative_traversal( root ):
2     stack = []
3     cur = root
4     preorders = []
5     inorders = []
6     while stack or cur:
7         while cur:
8             preorders.append( cur.val )
9             stack.append( cur )
10            cur = cur.left
11        node = stack.pop()
12        inorders.append( node.val )
13        cur = node.right
14    return preorders, inorders

```

11.4.3 Breath-first Tree Traversal

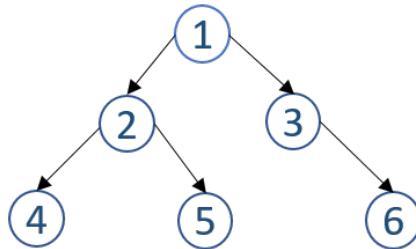


Figure 11.18: Draw the breath-first traversal order

Instead of traversing the tree recursively deepening down each time, the alternative is to visit nodes level by level, as illustrated in Fig. ?? for our exemplary binary tree. We first visit the root node 1, and then its children 2 and 3. Next, we visit 2 and 3's children in order, we goes to node 4, 5, and 6. This type of Level Order Tree Traversal uses the **breath-first search strategy** which differs from our covered depth-first search strategy. As we see in the example, the root node is expanded first, then all successors of the root node are expanded next, and so on, following a level by level ordering. We can also find the rule, the nodes first come and get first expanded. For example 2 is first visited and then 3, thus we expand 2's children first. Then we have 4 and 5. Next, we expand 3's children. This First come first expanded tells us we can rely on a queue to implement BFS.

Simple Implementation We start from the root, say it is our first level, put it in a list named `nodes_same_level`. Then we use a `while` loop, and each loop we visit all children nodes of `nodes_same_level` from the last level. We put all these children in a temporary list `temp`, before the loop ends, we assign `temp` to `nodes_same_level`, until the deepest level where no more children nodes will be found and leave our `temp` list to be empty and our while loop terminates.

```

1 def LevelOrder(root):
2     if not root:
3         return
4     nodes_same_level = [root]
5     while nodes_same_level:
6         temp = []
7         for n in nodes_same_level:
8             print(n.val, end=' ')
9             if n.left:
10                 temp.append(n.left)
11             if n.right:
12                 temp.append(n.right)
13         nodes_same_level = temp
  
```

The above will output follows with our exemplary binary tree:

```
1 1 2 3 4 5 6
```

Implementation with Queue As we discussed, we can use a FIFO queue to save the nodes waiting for expanding. In this case, at each `while` we only handle one node that are at the front of the queue.

```
1 def bfs(root):
2     if not root:
3         return
4     q = [root]
5     while q:
6         node = q.pop(0) # get node at the front of the queue
7         print(node.val, end=' ')
8         if node.left:
9             q.append(node.left)
10        if node.right:
11            q.append(node.right)
```

11.5 Informed Search Strategies**

11.5.1 Best-first Search

Best-first search is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule. The degree of promising of a node is described by a **heuristic evaluation function** $f(n)$ which, in general, may depend on the description of the node n , the description of the goal, and the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain.

Breath-first search fits as a special case in Best-first search if the objective of the problem is to find the shortest path from source to other nodes in the graph; it uses the estimated distance to source as a heuristic function. At the start, the only node in the frontier set is the source node, expand this node and add all of its unexplored neighboring nodes in the frontier set and each comes with distance 1. Now, among all nodes in the frontier set, choose the node that is the most promising to expand. In this case, since they all have the same distance, expand any of them is good. Next, we would add nodes that have $f(n) = 2$ in the frontier set, choose any one that has smaller distance.

A Generic best-first search will need a priority queue to implement instead of a FIFO queue used in the breath-first search.

11.5.2 Hands-on Examples

Get a more straightforward example

Add an example

Triangle (L120)

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

Example :

Given the following triangle :

```
[
[2],
[3,4],
[6,5,7],
[4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., $2 + 3 + 5 + 1 = 11$).

Analysis Solution: first we can use dfs traverse as required in the problem, and use a global variable to save the minimum value. The time complexity for this is $O(2^n)$. When we try to submit this code, we get LTE error. The code is as follows:

```

1 import sys
2 def min_path_sum(t):
3     """
4         Purely Complete Search
5     """
6     min_sum = sys.maxsize
7     def dfs(i, j, cur_sum):
8         nonlocal min_sum
9         # edge case
10        if i == len(t) or j == len(t[i]):
11            # gather the sum
12            min_sum = min(min_sum, cur_sum)
13            return
14        # only two edges/ choices at this step
15        dfs(i+1, j, cur_sum + t[i][j])
16        dfs(i+1, j+1, cur_sum + t[i][j])
17    dfs(0, 0, 0)
18    return min_sum

```

11.6 Exercises

11.6.1 Coding Practice

Property of Graph

1. 785. Is Graph Bipartite? (medium)
2. 261. Graph Valid Tree (medium)
3. 797. All Paths From Source to Target(medium)

12

Combinatorial Search

So far, we have learned the most fundamental search strategies on general data structures such as array, linked list, graph, and tree. In this chapter, instead of searching on explicit and well defined data structures, we extend and discuss more *exhaustive search* algorithms that can solve rather obscure and challenging *combinatorial problems*, such as sudoku and the famous Travels Salesman Problem. For combinatorial problems, we have to figure out the potential search space, and rummage a solution.

12.1 Introduction

Combinatorial search problems consists of n items and a requirement to find a solution, i.e., a set of $L < N$ items that satisfy specified conditions or constraints. For example, a sudoku problem where a 9×9 grid is partially filled with number between 1 and 9, fill the empty spots with numbers that satisfy the following conditions:

1. Each row has all numbers form 1 to 9.
2. Each column has all numbers form 1 to 9.
3. Each sub-grid (3×3) has all numbers form 1 to 9.

This sudoku together with one possible solution is shown in Fig. 12.1. In this case, we have 81 items, and we are required to fill 51 empty items with the above three constraints.

5	3			7				
6			1	9	5			
	9	8				6		
8			6				3	
4		8		3			1	
7			2			6		
	6				2	8		
		4	1	9			5	
		8			7	9		

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 12.1: A Sudoku puzzle and its solution

Model Combinatorial Search Problems We can model the combinatorial search solution as a vector $s = (s_0, s_1, \dots, s_{L-1})$, where each variable s_i is selected from a finite set A , which is called the *domain* for each variable. Such a vector might represent an arrangement where s_i contains the i -th item of a permutation, in the combination problem, a boolean denotes if the i -th item is selected already, or it can represent a path in a graph or a sequence of moves in a game. In the sudoku problem, each s_i can choose from a number in range $[1, 9]$.

Problem Categories Combinatorial search problems arise in many areas of computer science such as artificial intelligence, operations search, bioinformatics, and electronic commerce. These problems typically involve finding a *grouping*, *ordering*, or *assignment* of a discrete, finite set of objects that satisfy given conditions or constraints. We introduce two well-studied types of problems that are more likely to be NP-hard and of at least exponential complexity:

1. Constraint Satisfaction Problems (CSP) are mathematical questions defined as a set of variables whose state must satisfy a number of constraints or limitations (mathematical equations or inequations), such as Sudoku, N-queen, map coloring, Crosswords, and so on. The size of the search space of CSPs can be roughly given as:

$$O(cd^L) \quad (12.1)$$

Where there are L variables, each with domain size d , and there are c constraints to check out.

2. Combinatorial optimization problems consist of searching for maxima or minima of an objective function F whose domain is a discrete but large configuration space. Some classic examples are:

- Travelling Salesman Problems (TSP): given position (x, y) of n different cities, find the shortest possible path that visits each city exactly once.
- Integer Linear Programming: maximize a specified linear combination of a set of integers X_1, \dots, X_n subject to a set of linear constraints each of the form:

$$a_1X_1 + \dots + a_nX_n \leq c \quad (12.2)$$

- Knapsack Problems: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Search Strategies From Chapter Discrete Programming, we have learned the basic enumerative combinatorics, including counting principles and knowledge on permutations, combinations, partitions, subsets, and subsequences. Combinatorial Search builds atop this subject, and together through different search strategies such as depth-first search and best-first search, it is able to enumerate the search space and find the solution(s) with necessary speedup methods. In this chapter, we only discuss about complete search and only acknowledge the existence of approximate search techniques.

Backtracking is a process of depth-first based search where it “builds” the search tree on the fly incrementally instead of having a tree/graph structure beforehand to search through. Backtracking fits to solve combinatorial search problems because:

1. It is space efficient for the usage of a DFS and the candidates are built incrementally and their validity to fit a solution is checked right away.
2. It is time efficient for that some partial candidates can be pruned if the algorithm believes that it will not lead to our final complete solution.

Because the ordering of variables s_0, \dots, s_{L-1} can potentially affect the size of the search space sometimes. Thus, backtracking search relies on one or more heuristics to select which variable to consider next. *Look-ahead* is one such heuristic that is preferably applied to check the effects of choosing a given variable to evaluate or to decide the order of values to give to it.

There are other Breath-first Search based strategies that might work better than backtracking, such as for combinatorial optimization problems, best-first branch and bound search might be more efficient than its depth-first counterpart.

Speedups The speedup methods are well studied in computer science, and we list two general ways to prune unqualified or unnecessary branches during the search of backtracking:

1. **Branch and Prune:** This method prunes the unqualified branches with constraints of the problems. This is usually applied to solve constraint restricted problems (CSPs).
2. **Branch and Bound:** This method prunes unnecessary branches via comparing an estimation of a partial candidate with a found global best solution. If the estimation states that the partial candidate will never lead us to a better solution, we cut off this branch. This technique can be applied to solve a general optimization problems, such as Travel Salesman Problems (TSP), knapsack problems, and so.

12.2 Backtracking

In this section, we first introduce the technique of backtracking, and then demonstrate it by implementing common enumerative combinatorics seen in Chapter Discrete Programming.

12.2.1 Introduction

Backtracking search is an exhaustive search algorithm(depth-first search) that systematically assigns all possible combinations of values to the variables and checks if these assignments constitute a solution. Backtracking is all about choices and consequences and it shows the following two properties:

1. **No Repetition and Completion:** It is a systematic generating method that enumerates all possible states exactly at most once: it will not miss any valid solution but avoids repetitions. If there exists “correct” solution(s), it is guaranteed to be found. This property makes it ideal for solving combinatorial problems where the search space has to be constructed and enumerated. Therefore, the worst-case running time of backtracking search is exponential with the length of the state (b^L , b is the average choice for each variable in the state).
2. **Search Pruning:** Along the way of working with partial solutions, in some cases, it is possible for us to decide if they will lead to a valid *complete solution*. As soon as the algorithm is confident to say the partial configuration is either invalid or nonoptimal, it abandons this *partial candidate*, and then “backtracks” (return to the upper level), and resets to the upper level’s state so that the search process can continue to explore the next branch for the sake of efficiency. This is called *search pruning* with which the algorithm ends up amortizedly

visiting each vertex less than once. This property makes backtracking the most promising way to solve CSPs and combinatorial optimization problems.

Solving sudoku problem with backtracking algorithm, each time at a level in the DFS, it tries to extend the last partial solution $s = (s_0, s_1, \dots, s_k)$ by trying out all 9 numbers at s_{k+1} , say we choose 1 at this step. It testifies the partial solution with the desired solution:

1. If the partial solution $s = (s_0, s_1, \dots, s_k, 1)$ is still valid, move on to the next level and work on trying out s_{k+2} .
2. If the partial solution is invalid and is impossible to lead to a complete solution, it “backtracks” to the last level and resets the state as $s = (s_0, s_1, \dots, s_k)$ so that it can try our other choices if there are some left(which in our example, we will try $s_{k+1} = 2$) or keep “backtracking” to even upper level.

The process should be way clearer once we have learned the examples in the following subsections.

12.2.2 Permutations

Given a list of items, generate all possible permutations of these items. If the set has duplicated items, only enumerate all unique permutations.

No Duplicates(L46. Permutations)

When there are no duplicates, from Chapter Discreet Programming, we know the number of all permutations are:

$$p(n, m) = \frac{n!}{(n - m)!} \quad (12.3)$$

where m is the number of items we choose from the total n items to make the permutations.

For example :

`a = [1, 2, 3]`

There are 6 total permutations :

```
[1, 2, 3], [1, 3, 2],
[2, 1, 3], [2, 3, 1],
[3, 1, 2], [3, 2, 1]
```

Analysis Let us apply the philosophy of backtracking technique. We have to build a state with length 3, and each variable in the state has three choices: 1, 2, and 3. The constraint here comes from permutation which requires that no two variables in the state will be having the same value. To build this

incrementally with backtracking, we start with an empty state $[]$. At first, we have three options, we get three partial results $[1]$, $[2]$, and $[3]$. Next, we handle the second variable in the state: for $[1]$, we can choose either 2 or 3, getting $[1, 2]$ and $[1, 3]$; same for $[2]$, where we end up with $[2, 1]$ and $[2, 3]$; for $[3]$, we have $[3, 1]$ and $[3, 2]$. At last, each partial result has only one option, we get all permutations as shown in the example. We visualize this incrementally building candidates in Fig. 12.2.

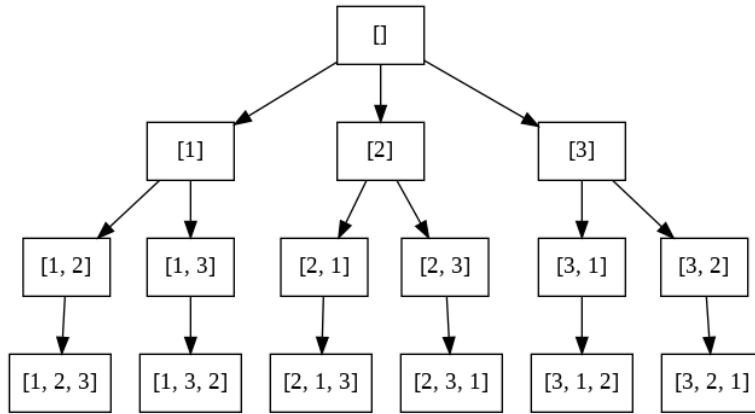


Figure 12.2: The search tree of permutation

However, we only managed to enumerate the search space, but not systematically or recursively with the Depth-first search process. With DFS, we depict the traverse order of the vertexes in the virtual search space with red arrows in Fig. 12.2. The backward arrows mark the “backtracking” process, where we have to reset the state to the upper level.

Implementation We use a list of boolean `bUsed` to track which item is used in the search process. `n` is the total number of items, `d` is the depth of the depth-first search process, `curr` is the current state, and `ans` is to save all permutations. The following code, we generate $p(n, m)$

```

1 def p_n_m(a, n, m, d, used, curr, ans):
2     if d == m: #end condition
3         ans.append(curr[:])
4         return
5
6     for i in range(n):
7         if not used[i]:
8             # generate the next solution from curr
9             curr.append(a[i])
10            used[i] = True
11            print(curr)
12            # move to the next solution
13            p_n_m(a, n, m, d + 1, used, curr, ans)
14            #backtrack to previous partial state
  
```

```

15     curr.pop()
16     used[i] = False
17     return

```

Check out the running process in the source code.

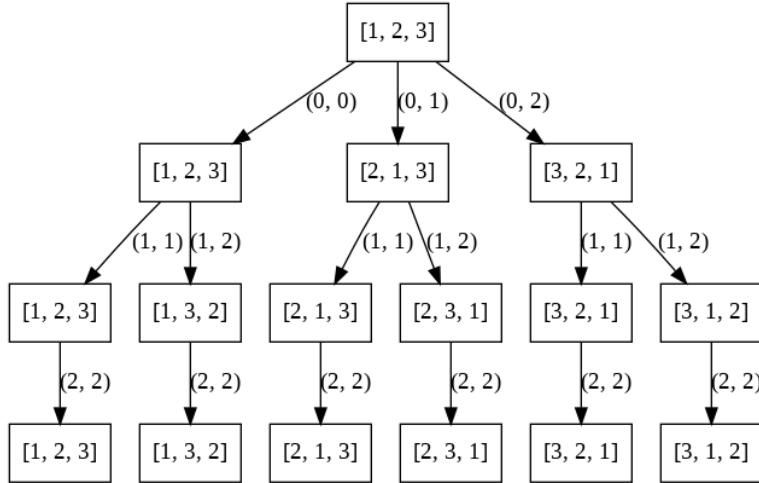


Figure 12.3: The search tree of permutation by swapping. The indexes of items to be swapped are represented as a two element tuple.

Alternative: Swapping Method We first start with a complete state, such that $s = [1, 2, 3]$ in our case. By swapping 1 and 2, we get $[2, 1, 3]$ and $[2, 3, 1]$ can be obtained by swapping 1 and 3 on top of $[2, 1, 3]$. With all permutations as leaves in the search space, the generating process is similar to Fig. 12.2. We show this alternative process in Fig. 12.3. At first, we swap index 0 with all other indexes, including 0, 1, and 2. At the second layer, we move on to swap index 1 with all other successive indexes, and so on for all other layers. The Python code is as:

```

1 ans = []
2 def permute(a, d):
3     global ans
4     if d == len(a):
5         ans.append(a[:])
6     for i in range(d, len(a)):
7         a[i], a[d] = a[d], a[i]
8         permute(a, d+1)
9         a[i], a[d] = a[d], a[i]
10    return

```

There is Johnson-Trotter algorithm that utilizes such swapping method, which avoids recursion, and instead computes the permutations by an iterative method.

With Duplicates(47. Permutations II)

We have already know that $p(n, n)$ is further decided by the duplicates within the n items. Assume we have in total of d items are repeated, and each item is repeated x_i times, then the number of all arrangements $pd(n, n)$ are:

$$pd(n, n) = \frac{p(n, n)}{x_0!x_1!...x_{d-1}}, \quad (12.4)$$

$$\text{w.r.t } \sum_{i=0}^{d-1} x_i \leq n \quad (12.5)$$

For example, when $a = [1, 2, 2, 3]$, there are $\frac{4!}{2!}$ unique permutations, which is 12 in total, and are listed as bellow:

```
[1 , 2 , 2 , 3] , [1 , 2 , 3 , 2] , [1 , 3 , 2 , 2] ,
[2 , 1 , 2 , 3] , [2 , 1 , 3 , 2] , [2 , 2 , 1 , 3] ,
[2 , 2 , 3 , 1] , [2 , 3 , 1 , 2] , [2 , 3 , 2 , 1] ,
[3 , 1 , 2 , 2] , [3 , 2 , 1 , 2] , [3 , 2 , 2 , 1]
```

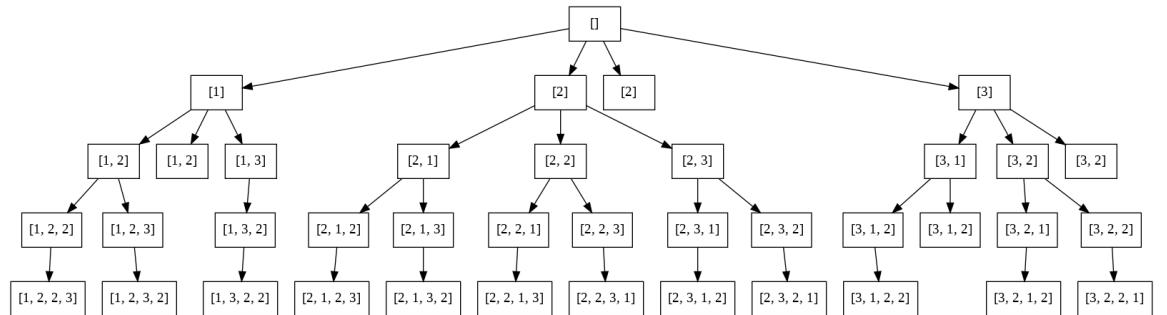


Figure 12.4: The search tree of permutation with repetition

Analysis The enumeration of these all possible permutations can be obtained with backtracking exactly the same as if there are no duplicates. However, this is not efficient since it has doubled the search space with repeated permutations. Here comes to our first time applying the Branch and Prune method: we avoid repetition by pruning off redundant branches.

One main advantage of backtracking is not to save all intermediate states, thus we should find a mechanism that avoids generating these intermediate states at the first place. One solution is that we sort all n items, making all repeat items adjacent to each other. We know if the current intermediate state is redundant by simply comparing this item with its predecessor: if it equals, we move on from building state with this item to the next item in line. The search tree of our example is shown in Fig. 12.4.

Implementation The implementation is highly similar to previous standard permutation code other than three different points:

1. Before the items are called by `permutate`, they are sorted first.
2. A simple condition check to avoid generating repeat states.
3. We used a dictionary data structure `tracker` which has all unique items as keys and each item's corresponding occurrence as values to replace the boolean vector `used` for slightly better space efficiency.

The Python code is as:

```

1 from collections import Counter
2 def permuteDup(nums, k):
3     ans = []
4     def permutate(d, n, k, curr, tracker):
5         nonlocal ans
6         if d == k:
7             ans.append(curr)
8             return
9         for i in range(n):
10            if tracker[nums[i]] == 0:
11                continue
12            if i - 1 >= 0 and nums[i] == nums[i - 1]:
13                continue
14            tracker[nums[i]] -= 1
15            curr.append(nums[i])
16
17            permutate(d+1, n, k, curr[:], tracker)
18            curr.pop()
19            tracker[nums[i]] += 1
20
21    return
22
23    nums.sort()
24    permute(0, len(nums), k, [], Counter(nums))
25    return ans

```



Can you extend the swap method based permutation to handle duplicates?

Discussion

From the example of permutation, we have demonstrated how backtracking works to construct candidates with an implicit search tree structure: the root node is the initial state, any internal node represents intermediate states, and all leaves are our candidates which in this case there are $n!$ for $p(n, n)$ permutation. In this subsection, we want to point out the unique properties and its computational and space complexities.

Two Passes Backtracking builds an implicit search tree on the fly, and it does not memorize any intermediate state. It visits the vertices in the search tree in two passes:

1. Forward pass: it builds the solution incrementally and reaches to the leaf nodes in a DFS fashion. One example of forward pass is $[] \rightarrow [1] \rightarrow [1, 2] \rightarrow [1, 2, 3]$.
2. Backward pass: as the returning process from recursion of DFS, it also backtracks to previous state. One example of backward pass is $[1, 2, 3] \rightarrow [1, 2], - \rightarrow [1]$.

First, the forward pass to build the solution **incrementally**. The change of `curr` in the source code indicates all vertices and the process of backtracking, it starts with `[]` and end with `[]`. This is the core character of backtracking. We print out the process for the example as:

```

[]->[1]->[1, 2]->[1, 2, 3]->backtrack: [1, 2]
backtrack: [1]
[1, 3]->[1, 3, 2]->backtrack: [1, 3]
backtrack: [1]
backtrack: []
[2]->[2, 1]->[2, 1, 3]->backtrack: [2, 1]
backtrack: [2]
[2, 3]->[2, 3, 1]->backtrack: [2, 3]
backtrack: [2]
backtrack: []
[3]->[3, 1]->[3, 1, 2]->backtrack: [3, 1]
backtrack: [3]
[3, 2]->[3, 2, 1]->backtrack: [3, 2]
backtrack: [3]
backtrack: []

```

Time Complexity of Permutation In the search tree of permutation in Fig. 12.2, there are in total V nodes, which equals to $\sum_{i=0}^n p_n^k$. Because in a tree the number of edges $|E|$ is $|v| - 1$, making the time complexity $O(|V| + |E|)$ the same as of $O(|V|)$. Since $p(n, n)$ itself alone takes $n!$ time, making the permutation an NP-hard problem.

Space Complexity A standard depth-first search consumes $O(bd)$ space in worst-case to execute, where b is branching factor and d is the depth of the search tree. In the combinatorial search problems, usually depth and branching is decided by the total number of variables in the state, making $b \sim d \sim n$. In backtracking, we have space complexity $O(n^2)$. However, in normal standard DFS, the input-tree or graph data structure—is given and not attributed to space complexity. For a NP-hard combinatorial search problem, this input is often exponential. Backtracking search outcompetes the standard DFS by avoiding such space consumption; it only keeps a dynamic data structure(`curr`) to construct node on the fly.

12.2.3 Combinations

Given a list of n items, generate all possible combinations of these items. If the input has duplicated items, only enumerate unique combinations.

No Duplicates (L78. Subsets)

From Chapter Discrete Programming, we list the powerset—all m -subset, $m \in [0, n]$ as:

$$C(n, m) = \frac{P(n, m)}{P(m, m)} = \frac{n!}{(n - m)!m!} \quad (12.6)$$

For example, when $a = [1, 2, 3]$, there are in total 7 m -subsets, they are:

```
C(3, 0): []
C(3, 1): [1], [2], [3]
C(3, 2): [1, 2], [1, 3], [2, 3]
C(3, 3): [1, 2, 3]
```

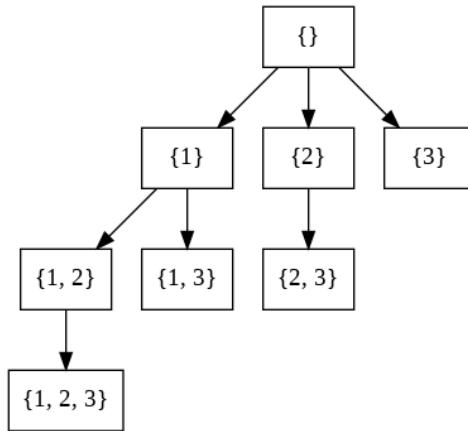


Figure 12.5: The Search Tree of Combination.

Analysis We can simply reuse the method of permutation, but with a problem that it generates lots of duplicates. For example, $P(3, 2)$ includes [1, 2] and [2, 1] which are indeed the same subset. Of course, we can check redundancy with saved m -subsets, but its not ideal. A systematical solution that avoids duplicates all along is preferred. If we limit the items we put into the m -subsets to be only increasing(of indexes of items or of values of items), in which case [2, 1], [3, 1], and [3, 2] will never be generated. The enumeration of combination through backtracking search is shown in Fig. 12.5.

Implementation Two modifications based on permutation code:

1. **for** loop: in the loop to iterate all possible candidates, we limit the candidates to be having larger indexes only.
2. We do not have to use a data structure to track the state of each candidate because any candidate that has larger index is a valid candidate.

We use **start** to track the starting position of valid candidates. The code of combination is:

```

1 def C_n_k(a, n, k, start, d, curr, ans):
2     if d == k: #end condition
3         ans.append(curr[:])
4         return
5
6     for i in range(start, n):
7         curr.append(a[i])
8         C_n_k(a, n, k, i+1, d+1, curr, ans)
9         curr.pop()
10    return

```

Alternative: 0 and 1 Selection We have discussed that a powerset written as $P(S)$. With each item either being appear or not appear in the resulting set makes the value set $\{0, 1\}$, resulting $|P(S)| = 2^n$. Follow this pattern, with our given example, we can alternatively generate a powerset like this:

```

s sets
1 {1}, {}
2 {1,2}, {1}, {2}, {}
3 {1,2,3}, {1,2}, {1, 3}, {3}, {2, 3}, {2}, {3}, {}

```

This process can be better visualized in a tree as in Fig. ???. We can see this process results 2^n leaves compared with our previous implementation which has a total of 2^n nodes is slightly less efficient. The code is as:

```

1 def powerset(a, n, d, curr, ans):
2     if d == n:
3         ans.append(curr[:])
4         return
5
6     # Case 1: select item
7     curr.append(a[d])
8     powerset(a, n, d + 1, curr, ans)
9     # Case 2: not select item
10    curr.pop()
11    powerset(a, n, d + 1, curr, ans)
12    return

```

Time Complexity The total nodes within the implicit search space of combination shown in Fig. 12.5 is $\sum_{k=0}^n C_n^k = 2^n$, which was explained in Chapter Discrete Programming. Thus, the time complexity of enumerating the powerset is $O(2^n)$ and is less compared with $O(n!)$ that comes with the permutation.

Space Complexity Similarly, combination with backtracking search uses slightly less space. But, we can still claim the upper bound to be $O(n^2)$.

With Duplicates(L90. Subsets II)

Assume we have m unique items, and the frequency of each is marked as x_i , with $\sum_{i=0}^{m-1} x_i = n$.

$$\sum_{k=0}^n c(n, k) = \prod_{i=0}^{m-1} (x_i + 1) \quad (12.7)$$

For example, when $a = [1, 2, 2, 3]$, there are $2 \times 3 \times 2 = 12$ combinations in the powerset, they are listed as below:

```
[] , [1] , [2] , [3] , [1, 2] , [1, 3] , [2, 2] , [2, 3] ,
[1, 2, 2] , [1, 2, 3] , [2, 2, 3] ,
[1, 2, 2, 3]
```

However, counting $c(n, k)$ with duplicates in the input replies on the specific input with specific distribution of these items. We are still able to count by enumerating with backtracking search.

Analysis and Implementation The enumeration of the powerset with backtracking search is the same as handling the iterations of choice in the enumeration of permutation with duplicates. We first sort our items in increasing order of the values. Then we replace the `for` loop from the above code with the following code snippet to handle the repetition of items from the input:

```
1 for i in range(start, n):
2     if i - 1 >= start and a[i] == a[i-1]:
3         continue
4     ...
```

12.2.4 More Combinatorics

In this section, we supplement more use cases of backtracking search in the matter of other types of combinatorics.

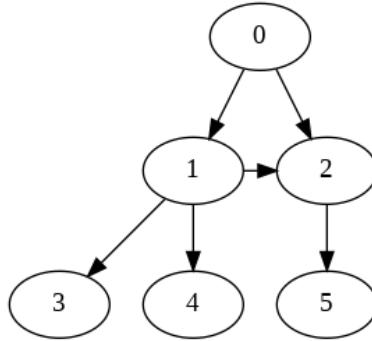


Figure 12.6: Acyclic graph

All Paths in Graph

For a given acyclic graph, enumerate all paths from a starting vertex s . For example, for the graph shown in Fig. 29.6, and a starting vertex 0, print out the following paths:

```
0, 0->1, 0->1->2, 0->1->2->5, 0->1->3, 0->1->4, 0->2, 0->2->5
```

Analysis The backtracking search here is the same as how to apply a DFS on an explicit graph, with rather one extra point: a state *path* which might have up to n items (the total vertices of a graph). In the implementation, the *path* vector will dynamically be modified to track all paths constructed as the go of the DFS. The code is offered as:

```

1 def all_paths(g, s, path, ans):
2     ans.append(path[:])
3     for v in g[s]:
4         path.append(v)
5         all_paths(g, v, path, ans)
6         path.pop()
  
```

You can run the above code in the Goolge Colab to see how it works on our given example.

Subsequences(940. Distinct Subsequences II)

Given a string, list all unique subsequences. There may or may not exist duplicated characters in the string. For example, when $s = '123'$, there are in total 7 subsequences, which are:

```
'', '1', '2', '3', '12', '13', '23', '123'
```

When $s = '1223'$ which comes with duplicates, there are 12 subsequences:

```
'', '1', '2', '3', '12', '13', '22', '23',
'122', '123', '223',
'1223'
```

Analysis From Chapter Discrete Programming, we have explained that we can count the number of unique subsequences through recurrence relation and pointed out the relation of subsequences with subsets(combinations). Let the number of unique subsequences of a sequence as $seq(n)$ and the number of unique subsets of a set as $set(n)$ with n items in the input. All subsequences are within subsets, and the subsequence set has larger cardinality than subsets, $|seq(n)| \geq |set(n)|$. From the above example, we can also see that when there are only unique items in the sequence or when there are duplicates but all duplicates of an item are adjacent to each other:

- The cardinality of subsequences and subsets equals, $|seq(n)| = set(n)|$.
- The subsequences and subsets share the same items when the ordering of the subsequences are ignored.

This indicates that the process of enumerating subsequences is almost the same as of enumerating a powerset. This should give us a good start.

Implementation However, if we change the ordering of the duplicated characters in the above string as $s = '1232'$, there are in total 14 subsequences instead:

```
'', '1', '2', '3', '12', '13', '23', '22', '32',
'123', '122', '132', '232',
'1232'
```

Therefore, our code to handle duplicates should differ from that of a powerset. In the case of powerset, the algorithm first sorts items so that all duplicates are adjacent to each other, making the checking of repetition as simple as checking the equality of item with its predecessor. However, in a given sequence, the duplicated items are not adjacent most of the time, we have to do things differently. We draw the search tree of enumerating all subsequences of string “1232” in Fig. 12.7. From the figure, we can observe that to avoid redundant branches, we simply check if a current new item in the subsequence is repeating by comparing it with all of its predecessors in range $[s, i]$. The code for checking repetition is as:

```
1 def check_repetition(start, i, a):
2     for j in range(start, i):
3         if a[i] == a[j]:
4             return True
5     return False
```

And the code to enumerate subsequences is:

```
1 def subseqs(a, n, start, d, curr, ans):
2     ans.append(''.join(curr[:]))
3     if d == n:
4         return
5
```

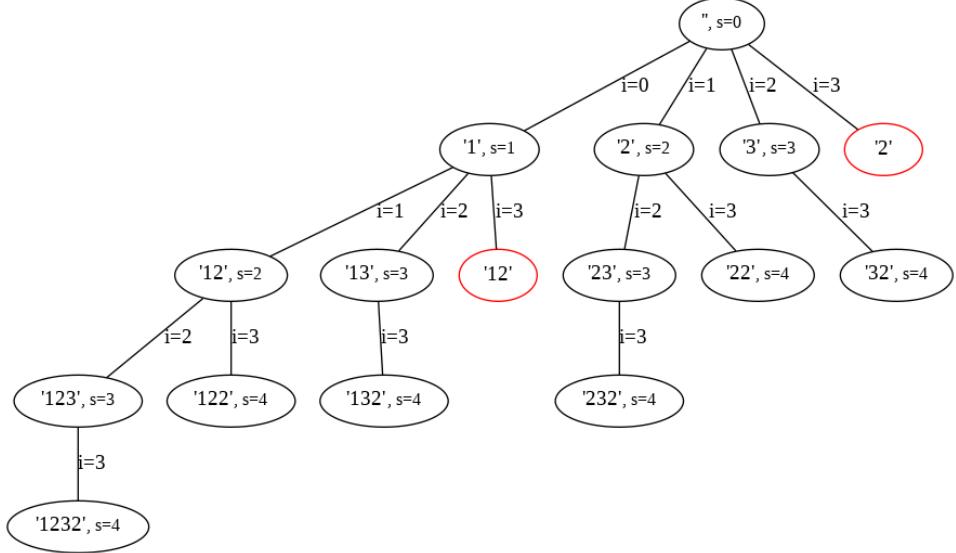


Figure 12.7: The Search Tree of subsequences. The red circled nodes are redundant nodes. Each node has a variable s to indicate the starting index of candidates to add to current subsequence. i indicate the candidate to add to the current node.

```

6   for i in range(start, n):
7       if check_repetition(start, i, a):
8           continue
9       curr.append(a[i])
10      subseqs(a, n, i+1, d+1, curr, ans)
11      curr.pop()
12  return

```

12.2.5 Backtracking in Action

So far, we have applied backtracking search to enumerate combinatorics. In this section, we shall see how backtracking search along with search pruning speedup methods solve two types of challenging NP-hard problems: Constraint Satisfaction Problems (CSPs) and Combinatorial Optimization Problems.

As we have briefly introduced the speedup methods needed to solve larger scale of CSPs and COPs. For example, assume within the virtual search tree, the algorithm is currently at level 2 with state $s = [s_0, s_1]$. If there are c choices for state s_1 , and if one choice is testified to be invalid, this will prune off $\frac{1}{c}$ of the whole search space. In this section, we demonstrate backtracking search armored with Branch and Prune method solving CSPs and Branch and Bound solving COPs.

12.3 Solving CSPs

Officially, a constraint satisfaction problem(CSP) consists of a set of n variables, each denoted as s_i , $i \in [0, n - 1]$; their respective value domains, each denoted as d_i ; and a set of m constraints, each denoted as c_j , $j \in [0, m - 1]$. A *solution* to a CSP is an assignment of values to all the variables such that no constraint is violated. A *binary* CSP is one in which each of the constraints involves at most two variables. A CSP can be represented by a *constraint graph* which has a node for each variable and each constraint, and an arc connecting variable nodes contained in a constraint to the corresponding constraint node.

We explain a few strategies from the CSP-solver's arsenal that can potentially speedup the process:

1. Forward Checking: The essential idea is that when a variable X from s_i is instantiated with a value x from its domain d_i , the domain of each future uninstantiated variable Y is examined. If a value y is found such that $X = x$ conflicts with $Y = y$, then y is temporarily removed from the domain of Y .
2. Variable Ordering: The order in which variables are considered while solving a CSP method can have a substantial impact on the search space. One effective ordering is always select the next variable with the smallest remaining domain. In a dynamic variable ordering, the order of variables is determined as the search progresses, and often goes with forward checking which keeps updating the uninstantiated variables' domains. Selecting variable with the minimal domain first can pinpoint the solution quickly given the fact that the branch is still early on, and branch pruning at this stage is more rewarding. Another reasoning is that each step, when we are multiplying d_i to the cost, we are adding the least expensive one, making this a greedy approach.

Sudoku (L37)

A Sudoku grid shown in Fig. 12.8 is a $n^2 \times n^2$ grid, arranged into n $n \times n$ mini-grids each containing the values $1, \dots, n$ such that no value is repeated in any row, column (or mini-grid).

Search Space First, we analyze the number of distinct states in the search space which relies on how we construct the intermediate states and our knowledge in Enumerative combinatorics. We discuss two different formulations on 9×9 grid:

1. For each empty cell in the puzzle, we create a set by taking values $1, \dots, 9$ and removing from it those values that appear as a given in the

5	3		7					
6			1	9	5			
	9	8				6		
8			6				3	
4		8		3				1
7			2			6		
	6				2	8		
		4	1	9				5
			8		7	9		

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 12.8: A Sudoku puzzle and its solution

same row, column, or mini-grid as that cell. Assume we have m spots and the corresponding candidate set of each spot is c_i , and initial cost estimation can be obtained which is:

$$T(n) = \prod_{i=0}^{m-1} c_i \quad (12.8)$$

2. Each row can be presented by a 9-tuples, there will be 9 rows in total, resulting 9 9-tuples to represent the search state. With c_i as the number of non-given values in the i -th 9-tuples, there are $c_i!$ ways of ordering these values by permuting. The number of different states in the search space is thus:

$$T(n) = \prod_{i=0}^8 c_i! \quad (12.9)$$

The two different ways each takes a different approach to formulate the state space, making its corresponding backtracking search differs too. We mainly focus on the first formulation with backtracking search.

Speedups Assume we have known all empty spots(variables) to fill in and we construct the search tree using backtracking. In our source code, we did an experiment comparing the effect of ordering variables with minimal domain first rule with arbitrary ordering. The experiment shows that the first method is more than 100 times faster than the second solving the our exemplary Sudoku puzzle. Therefore, we decide to always select the variable that has the least domain set to proceed next in the backtracking.

Further, we apply forward checking, for the current variable and a value we are able to assign, we recompute all the remaining empty spots' domain sets, and use the updated domain sets to decide:

- If this assignment will lead to empty domain for any of other remaining spots, and if so, we terminate the search and backtrack.

- The spot to select next time with the ordering rule we choose.

Implementation We set aside three vectors of length 9, `row_state`, `col_state`, and `block_state` to track the state of all 9 rows, columns, and grids. The list has `set()` data structures as items, saving the numbers filled already in that row, col, and grid respectively. Two stages in the implementation:

1. Initialization: We scan the whole each spot in the 9×9 grid to record the states of the filled spots and to find all empty spots that waiting to be filled in. With (i, j) to denote the position of a spot, it corresponds to `row_state[i]`, `col_state[j]`, and `block_state[i//3][j//3]`. We also write two functions to set and reset state with one assignment in the backtracking. The Python code is as follows:

```

1 from copy import deepcopy
2 class Sudoku():
3     def __init__(self, board):
4         self.org_board = deepcopy(board)
5         self.board = deepcopy(board)
6
7     def init(self):
8         self.A = set([i for i in range(1,10)])
9         self.row_state = [set() for i in range(9)]
10        self.col_state = [set() for i in range(9)]
11        self.block_state = [[set() for i in range(3)] for i in
12                           range(3)]
13        self.unfilled = []
14
15        for i in range(9):
16            for j in range(9):
17                c = self.org_board[i][j]
18                if c == 0:
19                    self.unfilled.append((i, j))
20                else:
21                    self.row_state[i].add(c)
22                    self.col_state[j].add(c)
23                    self.block_state[i//3][j//3].add(c)
24
25    def set_state(self, i, j, c):
26        self.board[i][j] = c
27        self.row_state[i].add(c)
28        self.col_state[j].add(c)
29        self.block_state[i//3][j//3].add(c)
30
31    def reset_state(self, i, j, c):
32        self.board[i][j] = 0
33        self.row_state[i].remove(c)
34        self.col_state[j].remove(c)
35        self.block_state[i//3][j//3].remove(c)

```

2. Backtracking search with speedups: In the initialization, we have another variable A used as the domain set of the current processing spot.

To get the domain set according to the constraints, a simple set operation is executed as: $A - (row_state[i] \cup col_state[j] \cup block_state[i//3][j//3])$. In the solver, each time, to pick a spot, we first update all remaining spots in the `unfilled` and then choose the one with minimal domain. This process takes $O(n)$ which is trivial compared with the cost of the searching, with 9 for computing domain set of a single spot, $9n$ for n spots, and adding another n to $9n$ to choose the one with the smallest size. The solver is implemented as:

```

1  def __ret_len(self, args):
2      i, j = args
3      option = self.A - (self.row_state[i] | self.col_state[j]
4                           | self.block_state[i//3][j//3])
5      return len(option)
6
7  def solve(self):
8      if len(self.unfilled) == 0:
9          return True
10     # Dynamic variables ordering
11     i, j = min(self.unfilled, key = self.__ret_len)
12     # Forward looking
13     option = self.A - (self.row_state[i] | self.col_state[j]
14                           | self.block_state[i//3][j//3])
15     if len(option) == 0:
16         return False
17     self.unfilled.remove((i, j))
18     for c in option:
19         self.set_state(i, j, c)
20         if self.solve():
21             return True
22         # Backtracking
23     else:
24         self.reset_state(i, j, c)
25     # Backtracking
26     self.unfilled.append((i, j))
27     return False

```

12.4 Solving Combinatorial Optimization Problems

Combinatorial optimization is an emerging field at the forefront of combinatorics and theoretical computer science that aims to use combinatorial techniques to solve discrete optimization problems. From a combinatorics perspective, it interprets complicated questions in terms of a fixed set of objects about which much is already known: sets, graphs, polytopes, and matroids. From the perspective of computer science, combinatorial optimization seeks to improve algorithms by using mathematical methods either to reduce the size of the set of possible solutions or to make the search itself faster.

Genuinely, the inner complexity of a COP is at least of exponential, and its solutions fall into two classes: exact methods and heuristic methods. In some cases, we may be able to find efficient exact algorithms with either greedy algorithms or dynamic programming technique such as finding the shortest paths on a graph can be solved by the Dijkstra (greedy) or Bellman-Ford algorithms(dynamic programming) to provide exact optimal solutions in polynomial running time. For more complex problems, COP can be mathematically formulated as a Mixed Linear Programming(MILP) model and which is generally solved using a linear-programming based branch-and-bound algorithm. But, in other cases no exact algorithms are feasible, and the following randomized heuristic search algorithms though we do not cover in this section should be applied:

1. Random-restart hill-climbing.
2. Simulated annealing.
3. Genetic Algorithms.
4. Tabu search.

Model Combinatorial Optimization Problems It is a good practice to formulate COPs with mathematical equations/inequations, which includes three steps:

1. Choose the decision variables that typically encode the result we are interested in, such that in a superset problem, each item is a variable, and each variable includes two decisions: take or not take, making its value set as 0, 1.
2. Express the problem constraints in terms of these decision variables to specify what the feasible solutions of the problem are.
3. Express the objective function to specify the quality of each solution.

There are generally many ways to model a COP.

Branch and Bound Branch and bound (BB, B&B, or BnB) is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it

cannot produce a better solution than the best one found so far by the algorithm. “Branching” is to split problem into a number of subproblems, and “bounding” is to find an optimistic estimation of the best solution to the subproblems to either maximize the upper bound or minimize the lower bound. To get the optimistic estimation, we have to *relax constraints*. In this section, we will exemplify both the minimization(TSP) and maximization problem(Knapsack).

Search Strategies In practice, we can apply different search strategies to enumerate the search space of the problem, such as depth-first, best-first, and least-discrepancy search. The way of how each listed strategy is applied in the combinatorial optimization problems is:

- Depth-First: it prunes when a node estimation is worse than the best found solution.
- Best-First: it selects the node with the best estimation among the frontier set to expand each time. Worst scenario, the whole search tree have to be saved as long the best estimation is extremely optimistic and not a single branch is pruned in the process.
- Least-Discrepancy: it trusts a greedy heuristic, and then move away from the heuristic in a very systematic fashion.

In this section, we discuss exact algorithms using Branch and Bound with a variation of search strategies. During the interviews, questions that have polynomial exact solutions are more likely to appear, proving your mastery of dynamic programming or greedy algorithms design methodologies. However, it is still good to discuss this option.

12.4.1 Knapsack Problem

Given n items with weight and value indicated by two vectors W and V respectively. Now, given a knapsack with capacity c , maximize the value of items selected into the knapsack with the total weight being bounded by c . Each item can be only used at most once. For example, given the following data, the optimal solution is to choose item 1 and 3, with total weight of 8, and optimal value of 80.

```
c = 10
W = [5, 8, 3]
V = [45, 48, 35]
```

Search Space In this problem, x_i denotes each item, and w_i, v_i for its corresponding weight and value, with $i \in [0, n - 1]$. Each item can either be selected or left behind, indicating $x_i \in \{0, 1\}$. The selected items can not

exceed the capacity, making $\sum_{i=0}^{n-1} w_i x_i \leq c$. And we capture the total value of the selected items as $\sum_{i=0}^{n-1} v_i x_i$. Putting it all together:

$$\max_{v,x} \quad \sum_{i=0}^{n-1} v_i x_i \tag{12.10}$$

$$\text{s.t.} \quad \sum_{i=0}^{n-1} w_i x_i \leq c \tag{12.11}$$

$$x_i \in \{0, 1\} \tag{12.12}$$

With each variable having two choices, our search space is as large as 2^n .

Branch and Bound To bound the search, we have to develop a heuristic function to estimate an optimistic–maximum–total value a branch can lead to.

In the case of knapsack problem, the simplest estimation is summing up the total values of selected items so far, and estimate the maximum value by adding the accumulated values of all remaining unselected items along the search.

A tighter heuristic function can be obtained with **constraint relaxation**. By relaxing the condition of simply choose $\{0, 1\}$ to $[0, 1]$, that a fraction of an item can be chosen at any time. By sorting the items by the value per unit $\frac{v_i}{w_i}$, then a better estimate can be obtained by filling the remaining capacity of knapsack with unselected items, with larger unit value first be considered. A branch is checked on the optimal solution so far against the lower estimated bound in our case, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm. Both heuristic functions are more optimistic compared with the true value, but the later is a tighter bound, being able to prune more branches along the search and making it more time efficient. We demonstrate branch and bound with two different search strategies: DFS(backtracking) and Best-First search.

Depth-First Branch and Bound

We set up a class `BranchandBound` to implement this algorithm. First, in the initiation, we add additional $\frac{v_i}{w_i}$ to mark each item's value per unit, and sort these items by this value in decreasing order. Second, we have a function `estimate` which takes three parameters: `idx` as start index of the remaining items, `curval` is the total value based on all previous decision, and `left_cap` as the left capacity of the knapsack. The code snippet is:

```

1 import heapq
2
3 class BranchandBound:

```

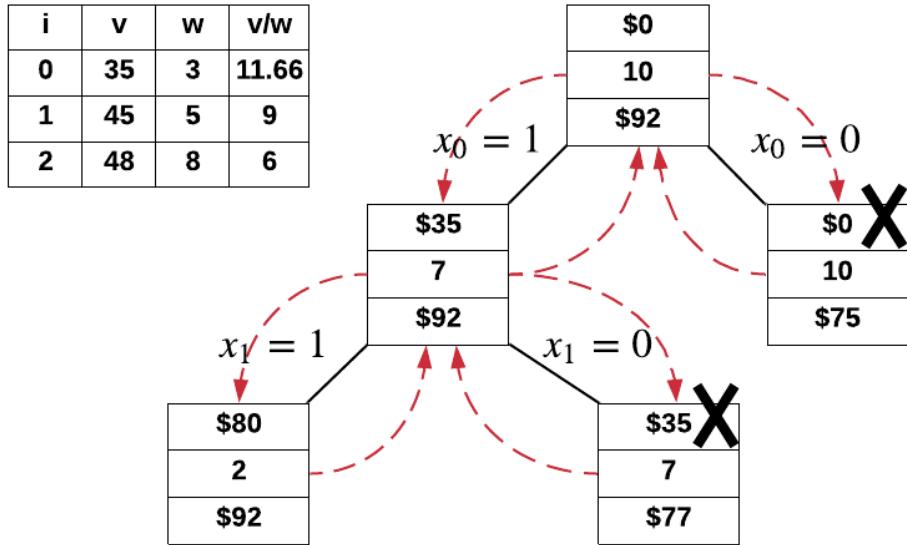


Figure 12.9: Depth-First Branch and bound

```

4     def __init__(self, c, v, w):
5         self.best = 0
6         self.c = c
7         self.n = len(v)
8         self.items = [(vi/wi, wi, vi) for _, (vi, wi) in enumerate(
9             zip(v, w))]
10        self.items.sort(key=lambda x: x[0], reverse=True)
11
12    def estimate(self, idx, curval, left_cap):
13        est = curval
14        # use the v/w to estimate
15        for i in range(idx, self.n):
16            ratio, wi, _ = self.items[i]
17            if left_cap - wi >= 0: # use all
18                est += ratio * wi
19                left_cap -= wi
20            else: # use part
21                est += ratio * (left_cap)
22                left_cap = 0
23        return est

```

In the Depth-first search process, it consists of two main branches: one considering to choose the current item, and the other to handle the case while the item is not selected. For the first branch, it has to be bounded by the capacity and the comparison of the best found solution against to the estimation. Additional `status` is to assist to visualize the process of the search, which tracks the combination of items. The process is shown in Fig. 12.9. And the code is as:

```
1     def dfs(self, idx, est, val, left_cap, status):
```

```

2     if idx == self.n:
3         self.best = max(self.best, val)
4         return
5     print(status, val, left_cap, est)
6
7     _, wi, vi = self.items[idx]
8     # Case 1: choose the item
9     if left_cap - wi >= 0: # prune by constraint
10        # Bound by estimate, increase value and volume
11        if est > self.best:
12            status.append(True)
13            nest = self.estimate(idx+1, val+vi, left_cap - wi)
14            self.dfs(idx+1, nest, val+vi, left_cap - wi, status)
15            status.pop()
16
17     # Case 2: not choose the item
18     if est > self.best:
19         status.append(False)
20         nest = self.estimate(idx+1, val, left_cap)
21         self.dfs(idx+1, nest, val, left_cap, status)
22         status.pop()
23     return

```

Best-First Branch and Bound

Within Best-First search, we use priority queue with the estimated value, and each time the one with the largest estimated value within the frontier set is expanded first. Similarly, with branch and bound, we prune branch that has estimated value that would never surpass the best solution up till then. The search space is the same as in Fig. 12.9 except that the search process is different from depth-first. In the implementation, the priority queue is implemented with a min-heap where the minimum value is firstly popped out, thus we use the negative estimated value to make it always pop out the largest value conveniently instead of write code to implement a max-heap.

```

1 def bfs(self):
2     # track val, cap, and idx is which item to add next
3     q = [(-self.estimate(0, 0, self.c), 0, self.c, 0)] #
4     estimate, val, left_cap, idx
5     self.best = 0
6     while q:
7         est, val, left_cap, idx = heapq.heappop(q)
8         est = -est
9         _, wi, vi = self.items[idx]
10        if idx == self.n - 1:
11            self.best = max(self.best, val)
12            continue
13
14        # Case 1: choose the item
15        nest = self.estimate(idx + 1, val + vi, left_cap - wi)
16        if nest > self.best:

```

```

16     heapq.heappush(q, (-nest, val + vi, left_cap - wi, idx
+ 1))
17
18     # Case 2: not choose the item
19     nest = self.estimate(idx + 1, val, left_cap)
20     if nest > self.best:
21         heapq.heappush(q, (-nest, val, left_cap, idx + 1))
22     return

```

12.4.2 Travelling Salesman Problem

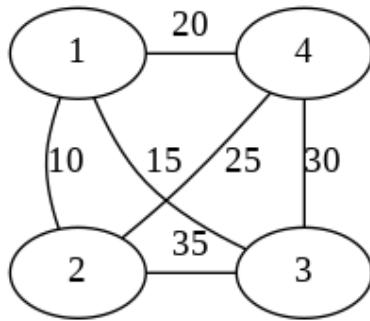


Figure 12.10: A complete undirected weighted graph.

Given a set of cities and the distances between every pair, find the shortest possible path that visits every city exactly once and returns to the origin city. For example, with the graph shown in Fig. 12.10, such shortest path is $[0, 1, 3, 2, 0]$ with a path weight 80.

Search Space In TSP, a possible complete solution is a *Hamiltonian cycle*, a graph cycle that visits each vertex exactly once. Since it is a cycle, it does not matter where it starts. For convenience, we choose vertex 0 as the origin city. Therefore, in our example, our path starts and ends at 0, and the remaining $n - 1$ vertices between will be a permutation of these vertices, making the complexity as $(n - 1)!$.

Because this is a complete graph, it might be tempting to apply backtracking on the graph to enumerate all possible paths and find and check possible solutions. However, this path searching will build a $n - 1$ -ary search tree with height equals to $n - 1$, making the complexity as $\frac{(n-1)^{n-1}}{n-2}$, which is larger than the space of permutation among $n - 1$ items. Therefore, in our implementation, we apply backtracking to enumerate all permutations of $n - 1$ vertices and check its corresponding cost.

Speedups Since we only care about the minimum cost, then any partial result that has cost larger than the minimum cost of all known complete

solutions can be pruned. This is the *Branch and bound* method that we have introduced that is often used in the combinatorial optimization.

Implementation We built the graph as a list of dictionaries, each dictionary stores the indexed vertex's other cities and its corresponding distance as key and value respectively. Compared with standard permutation with backtracking, we add four additional variables: `start` to track the starting vertex, `g` to pass the graph to refer the distance information, `mincost` to save the minimum complete solution so far found, and `cost` to track the current partial path's cost. The code is shown as:

```

1 def tsp(a, d, used, curr, ans, start, g, mincost, cost):
2     if d == len(a):
3         # Add the cost from last vertex to the start
4         c = g[curr[-1]][start]
5         cost += c
6         if cost < mincost[0]:
7             mincost[0] = cost
8             ans[0] = curr[::-1] + [start]
9         return
10
11    for i in a:
12        if not used[i] and cost + g[curr[-1]][i] < mincost[0] :
13            cost += g[curr[-1]][i]
14            curr.append(i)
15            used[i] = True
16            tsp(a, d + 1, used, curr, ans, start, g, mincost, cost)
17            curr.pop()
18            cost -= g[curr[-1]][i]
19            used[i] = False
20    return

```

TSP is a NP-hard problem, and there is no known polynomial time solution so far.

Other Solutions

Whenever we are faced with optimization, we are able to consider the other two algorithm design paradigm—Dynamic Programming and Greedy Algorithms. In fact, the above two problems both have its corresponding dynamic programming solutions: for knapsack problem, polynomial solution is possible; for TSP, though it is still of exponential time complexity, it is much better than $O(n!)$. We will further discuss these two problems in Chapter Dynamic Programming.

12.5 Exercises

1. 77. Combinations

2. 17. Letter Combinations of a Phone Number
3. 797. All Paths From Source to Target
4. N-bit String: enumerate all n-bit strings with backtracking algorithm, for example:

```
n = 3, all 3-bit strings are:  
000, 001, 010, 011, 100, 101, 110, 111
```

5. 940. Distinct Subsequences II
6. N-queen
7. Map-coloring
8. 943. Find the Shortest Superstring (hard). Can be moduled as traveling salesman problem and dynamic programming

13

Reduce and Conquer

“Everything should be made as simple as possible, but not simpler.”
— Albert Einstein,

This chapter is the essence of the algorithmic problem solving—‘Reduction’.

Reduction is the essence of problem solving, and self-reduction is the “center” of the essence. Recurrence relation is our tool from math. The correctness of self-reduction is proved with mathematical induction. And the complexity analysis relies on solving recurrence relation.



13.1 Introduction

Story Imagine that your mom asks you to get 10,000 pounds of corns, what would you do? First, you would think, where should I get the corns? I can go to Walmart or I can go to grow the corns in the farm. This is when one problem/task is reduced to some other problems/tasks. Solving the other ones means you solved your assignment from your mom. This is one example of the reduction; converting problem *A* to problem *B*.

Now, you are at Walmart and are ready to load the 10,000 pounds of bagged corns, but the trunk of your car can not fit all corns at once. You just decide that you want to do 10 rounds of loading and transporting to home. Now, your task becomes loading 1,000 pounds of corns. After you are done with this, you just solved a subtask—getting 1,000 pounds of corns. In the second round, you load another 1,000 pounds. You solved another subtask—getting 2,000 pounds of corns. After 10 rounds in total, you will solve the

original task. This is the other side of reduction, reduce one problem to one or multiple smaller instances of itself.

Definition of Reduction In computational theory and computational complexity theory, a reduction is an algorithm for transforming one problem A into another problem or other problems. There are two types of reduction:

1. **Self-Reduction:** it can also be a problem that are just a smaller instance or we say *subproblems* of itself, say if the original problem is a_n , then the smaller problems can be $a_{n/2}, a_{n-1}, a_{n-2}$, and so on. Self-reduction is a recursive process; we reduce the problem into one or more subproblems of smaller size recursively until the subproblem is small enough to be a base case. We need to differentiate whether the subproblem is reduced by constant factor or just by constant size.
 - If it is by constant size, say a_{n-k} , this will characterize *searching*, *dynamic programming* and *greedy algorithm*.
 - If it is by constant factor, say in the form of $a_{n/b}$, b is integer $b \geq 2$, this can be used to characterize *divide and conquer* which we detail on it further in Section. 13.2.1.

The **Recurrence relations** which we have put so much effort on in last chapter will conveniently represent and interpret the relation between problem and its subproblems in self-reduction. Optionally, we can also use *recursion tree* to help with visualization. In the next two sections, we shall see how and discuss additional techniques for each type.

2. **A to B :** The other problem can be a totally different problem, say B . Intuitively, if we know how to solve B , this induces a solution to A . On the other hand, this also means that if any of A and B is unsolvable, then it indicates or proves that the other is unsolvable. More details will be given in Section. 13.5.

Reducing a Problem to Subproblem(s)

“Reducing” a problem into subproblem(s) as the first step of using self-reduction will result potentially two types of subproblems: *non-overlapping subproblems* and *overlapping subproblems*.

Non-overlapping subproblems Like cutting a rod into multiple pieces, the resulting subproblems each stand alone, disjoint with each other and become another rod which is just smaller. The most general way is to divide equally, thus conventionally $a_{n/b}$ means the problem is reduced into non-overlapping subproblems and each with size n/b .

Overlapping subproblems Different from the non-overlapping problems, the feature of overlapping problem is more abstract. Easily put, it means subproblems share subproblems. Say a_n is reduced to a_{n-1} and a_{n-2} , and according to this recursive rule, a_{n-1} will be reduced to a_{n-2} and a_{n-3} . Now we can see that problem a_n and a_{n-1} both share a common subproblem a_{n-2} , this is to say that these problems might overlap. Overlapping subproblems is one of the signals that further optimization might apply, which is detailed in Dynamic programming in Chapter. 15.

Self-Reduction and Mathematical Induction

The word ‘self-Reduction’ is not commonly or even put under the umbrella of ‘reduction’. In other materials, you might see that the content of self-reduction appears in the form of mathematical induction¹. Self-Reduction and Mathematical Induction are inseparable, as self-reduction can be represented with recurrence relation, and the mathematical induction is the most straightforward and powerful tool to prove its correctness and their concentration aligns—“concentrating on reducing a problem and solving subproblems rather than solving it directly”.

Mathematical induction can guide us to reduce the problem: we assume we know the solution from problems of size $a_{n/b}$, or a_{n-k} , we focus on how to construct a solution for a_n with solutions to our subproblems such as $a_{n/b}$ and a_{n-k} .

We will further see the distinction of these two characteristics of problems in our following examples.

13.2 Divide and Conquer

13.2.1 Concepts

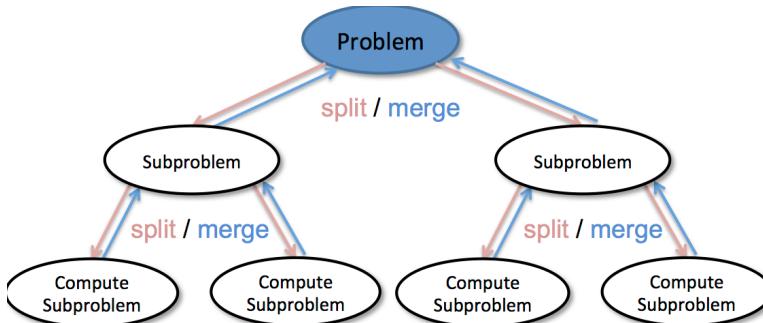


Figure 13.1: Divide and Conquer Diagram

¹Such as *Introduction to Algorithms: A Creative Approach*.

Divide and conquer is the most fundamental problem solving paradigm for computer programming; the strategy is to divide a problem into smaller problems recursively until the subproblem is trivial to solve. In more details, it consists of two process:

1. **Divide:** divide one problems into a series of non-overlapping subproblems that are smaller instances of the same problem recursively until reaching to the *bases cases*, when the subproblem is trivial to solve. Usually, the problem is divided equally, and most likely breaking into half and half. We say a problem of size n , denote as p_n is divided into a subproblems and each with size n/b , denote as $ap_{n/b}$, a, b are mostly integer and $a \geq 1, b \geq 2$. As we explained in Chapter. III, this process happens at the top-down pass.
2. **Conquer:** this step means that in the bottom-up pass, say we have the solutions of the a subproblems each with size n/b available, we need to *combine* these solutions to our current problem of size n .

We can interpret the divide and conquer with a recurrence relation as in Eq. 13.1

$$p_n = \Psi(n, ap_{n/b}) \quad (13.1)$$

Ψ will no longer be a function any more, but instead it represents the operations needed to combine the the solutions to subproblems to solution of current problem, n means the size of the combined solutions will be mostly n , which also means n elements.

Decrease and Conquer When $a = 1$, each problem reduced to only one sub-problem, and this case is named as **Decrease and Conquer**. Decrease and conquer will reduce search space each step. If our time complexity is $T(n) = T(n/2) + O(1)$, we get $O(\log n)$. This decrease and conquer method cuts the search space into half of its original at each step until it reaches its target. Because logarithmic is way faster even compared with linear, this is a significant efficiency growth. We will discuss classical algorithms with this paradigm such as Binary Search, Binary Search Tree, Segment Tree in the next chapter.

Common Applications of Divide and Conquer

Divide-and-conquer is mostly used in some well-developed algorithms and some data structures. In this book, we covered the follows:

- Various sorting algorithms like Merge Sort, Quick Sort (Chapter 15);
- Binary Search (Section ??);
- Heap(Section ??);

- Binary Search Tree (Section 27.1);
- Segment Tree(Section 27.2).

13.2.2 Hands-on Examples

Merge Sort

The concept can be quite dry, let us look at a simple example of merge sort. Given an array, [2, 5, 1, 8, 9], the task is to sort the array to [1, 2, 5, 8, 9]. To apply divide and conquer, we first divide it into two halves: [2, 5, 1], [8, 9], sort each half and with return result [1, 2, 5], [8, 9], and now we just need to merge the two parts. The process can be represented as the following:

```

1 def divide_conquer(A, s, e):
2     # base case, can not be divided farther
3     if s == e:
4         return A[s]
5     # divide into n/2, n/2 from middle position
6     m = (s+e) // 2
7
8     # conquer
9     s1 = divide_conquer(A, s, m)
10    s2 = divide_conquer(A, m+1, e)
11
12    # combine
13    return combine(s1, s2)
```

This process can be visualized in Fig. 13.2. From the visualization, we can clearly see that all subproblems form a tree and they never interact or overlap with each other, and each subproblem will only be visited once.

Maximum Subarray (53. medium).

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [-2, 1, -3, 4, -1, 2, 1, -5, 4],
the contiguous subarray [4, -1, 2, 1] has the largest sum = 6.

Solution: divide and conquer. $T(n) = \max(T(left), T(right), T(cross))$), max is for merging and the T(cross) is for the case that the potential subarray across the mid point. For the complexity, $T(n) = 2T(n/2) + n$, if we use the master method, it would give us $O(nlgn)$. We write the following Python code

```

1 def maxSubArray(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: int
5     """
6     def getCrossMax(low, mid, high):
```

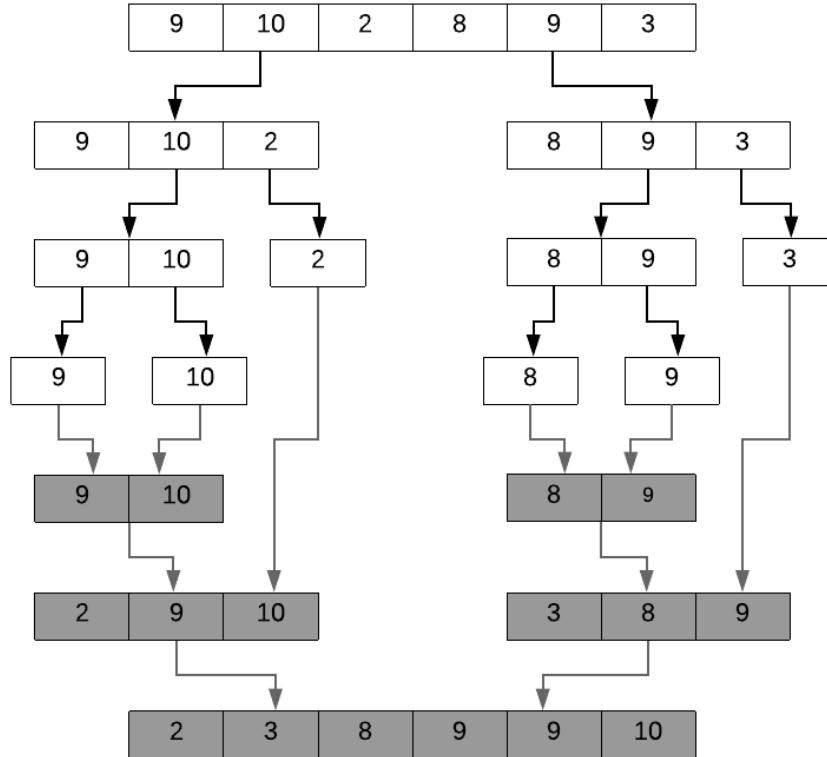


Figure 13.2: Merge Sort with non-overlapping subproblems where subproblems form a tree

```

7         left_sum,right_sum =0,0
8         left_max, right_max = -maxint, -maxint
9         left_i ,right_j=-1,-1
10        for i in xrange(mid,low-1,-1): #[]
11            left_sum+=nums[ i ]
12            if left_sum>left_max:
13                left_max= left_sum
14                left_i = i
15        for j in xrange(mid+1,high+1):
16            right_sum+=nums[ j ]
17            if right_sum>right_max:
18                right_max= right_sum
19                right_j = j
20        return (left_i ,right_j ,left_max+right_max)
21
22    def maxSubarray(low ,high):
23        if low==high:
24            return (low ,high , nums[ low ])
25        mid = (low+high)//2
26        rslt=[]
27        #left_low , left_high , left_sum = maxSubarray(low ,mid

```

```

28     ) #[low ,mid]
29         rslt .append( maxSubarray( low ,mid)) #[ low ,mid]
30         #right_low ,right_high ,right_sum = maxSubarray( mid+1,
31         high )#[mid+1,high ]
32             rslt .append( maxSubarray( mid+1,high ))
33             #cross_low ,cross_high ,cross_sum = getCrossMax( low ,
34             mid , high )
35                 rslt .append( getCrossMax( low , mid , high ))
36                 return max( rslt , key=lambda x: x[2])
37             return maxSubarray( 0 ,len (nums)-1 )[2]

```

13.3 Constant Reduction

13.3.1 Concepts

In this category, problem instance of size n is reduced to one or more instances of size $n - 1$ or less recursively until the subproblem is small and trivial to solve. This process can be interpreted with Eq. 13.2.

$$p_n = \Psi(n, p_{n-1}, p_{n-2}, \dots, p_{n-k}) \text{ for } n \leq k, \quad (13.2)$$

The number of subproblems that a current problem relies on should be as less as possible. The ideal option is when it only relates to $n - 1$, this is the case of an exhaustive search, which can be implemented easily both with recursion and iteration.

Overlapping Subproblems

When the number of the subproblems appears in this relation is larger or equals to 2, the subproblems might overlap. This implies that a straightforward recursion based solution without optimization will be expensive because these overlapped problems are solved again and again; the optimization is possible with dynamic programming or greedy algorithm shown in Part. ?? which optimize it using *caching mechanism* by saving the solution of each subproblem and thus avoiding recomputation. However, to stick to just the reduction itself, we delay our examples' possible optimization to Part. ??.

Subproblem Space

To count all possible subproblems— the subproblem space—is important for us to understand the complexity. For array, a subproblem can be a subarray that $[a_i, \dots, a_j], i < j, i \in [0, n - 1], j \in [0, n - 1]$, which makes the potential subproblems n^2 . Sometimes, it is enough to fix $a_i = a_0$, that the subarray always start from the start. The reduction by constant size will be less likely to be seen in tree structure where its more organized by the divide

and conquer reduction. In graph, this type of reduction??? In practice, we shall always try to define our subproblems that comes with smallest possible subproblem space, we would only enlarge it when we feel the further shrinking will make the construction of the solution impossible.

13.3.2 Hands-on Examples

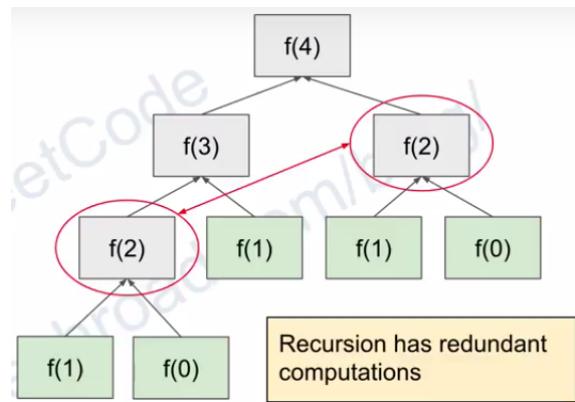


Figure 13.3: Fibonacci number with overlapping subproblems where subproblems form a graph.

Example 2: Fibonacci Sequence The Fibonacci Sequence is defined as:

Given $f(0)=0$, $f(1)=1$, $f(n) = f(n-1) + f(n-2)$, $n \geq 2$. Return the value for any given n .

The above is the classical Fibonacci Sequence, to get the fibonacci number at position n , we first need to know the answer for subproblems $f(n-1)$ and $f(n-2)$, we can solve it easily using recursion function:

```

1 def fib(n):
2     if n <= 1:
3         return n
4     return fib(n-1) + fib(n-2)

```

The above recursion function has recursion tree shown in Fig ???. And we also draw the recursion tree of recursion function call for merge sort and shown in Fig 13.2. We notice that we call $f(2)$ multiple times for fibonacci but in the merge sort, each call is unique and wont be called more than once. The recurrence function of merge sort is $T(n) = 2 * T(n/2) + n$, and for fibonacci sequence it is $T(n) = T(n - 1) + T(n - 2) + 1$.

Maximum Subarray Using reduction by constant size, the problem is to find a subarray a_i, a_{i+1}, \dots, a_j , $i \geq 0, i \leq j < n$. Now, let's assume that the

solution of subarray of size $n - 1$ is known with the induction hypothesis, try to figure out the “operations” to construct the solution for n .

For subproblem of size 4, it is $[-2, 1, -3]$, where is the maximum subarray? A naive and human doable way is to check all subarrays, which will be

```
subarray start from -2: -2, -1, -4
subarray start from 1: 1, -2
subarray start from -3: -3
```

we would find $[1]$ is our maximum subarray, to subproblem $[-2, 1, -3, 4]$, to construct the solution, we have two cases:

1. $j = n - 1$, to put 4 inside, we just need to make a choice in three cases: previous maximum subarray $[a_i, \dots, a_{n-1}]$, extended maximum subarray $[a_i, \dots, a_{n-1}, a_n]$, and $[a_n]$.
2. $j \neq n - 1$, in this case, say our maximum subarray is $[a_i, \dots, a_j], j \neq n - 1$, the case of extension is more complex, that we need to try extend any item in $[a_{j+1}, a_n]$. Its doable but gives us time complexity of $T(n) = T(n - 1) + O(n)$, this is the same as of in the naive solution to enumerate all subarrays.

13.4 Divide-and-conquer VS Constant Reduction

Therefore, we draw the conclusion:

1. For non-overlapping as Eq. ??, when we use recursive programming to solve the problem directly, we get the best time complexity since there is no overlap between subproblems.
2. For overlapping problems as Eq. ??, programming them recursively would end up with redundancy in time complexity because some subproblems are computed more than one time. This also means they can be further optimized: either using recursive with memoization or iterative through tabulation as we later explain in Chapter **Dynamic Programming** (Chapter 15).

Practical Guideline With the master theorem to either divide and conquer or reducing by constant size tells us its asymptotic time complexity: divide and conquer is either polynomial or logarithmic and reducing by constant size will go up to exponential with $f(n) \geq n$. This reminds us that when our problem state space is beyond exponential, we might better try reducing by constant size, and if the problem state space is within polynomial, a divide and conquer should work better to further boost the efficiency.

13.5 A to B

13.5.1 Concepts

Definition Reduction or transformation between two problems A and B is to say that a solution of one problem is also a solution of the other. It's common applications are:

1. Design algorithms: given algorithm for B , can solve our problem for A .
2. Proving limits: If A is hard that its lower bound is known, then so is B .
3. Classifying problems using the established relative difficulty of problems.

The strategy is to reduce problem A into a more general-purposed problem. One of such general-purposed problem is *linear programming*, which we study in Chapter. ???. Only experience and study of the classical algorithms designed on a certain data structures can help us identify possible ‘patterns’. In this section, we emphasize the principle of this method instead of pointing out all possible reduction; as we are further in the book and the journey of practice, we will learn.

13.5.2 Practical Guideline and Examples

Sorting and Ordering is always a good try. We only focus on Design algorithms in this book with this method. In practice, *sorting* or *ordering* is a common trick to reduce a problem to another that is more structured and with well-known algorithms to solve. For example, if we need to find the k -th largest item in an array, sorting the array first serves a more general purpose and solves the problem. Same with the problem that we are given an unsorted array, find if there exist duplicate items. This case fully demonstrated how a well-known and a general purposed sorting algorithm can be used to solve a problem that might seems bizarre or unique.

However, more likely sorting will be a subroutine of our algorithm design, whose purpose is to order our input in the hope that it simplifies the following design. Similar examples can be found across the book, other example will be in the convex hull problem where the points are sorted by angle to outskirt chosen point. The real-world is fuzzy, as a computer scientist, we are responsible to enforce orders, so always keep this in mind.

Other Reduction Examples There are also more creative reduction, as we see in the last section of example maximum subarray, we were having a

problem to use straightforward induction hypothesis, we can strengthen our hypothesis.

If we reduce our problem to be: find the maximum subarray that ends at n , that is find $[a_i, \dots, a_{n-1}], i \geq 0$ in the array that has the maximum value. We simply only have n candidates to compare. To construct this reduced problem B back to A, the maximum subarray of A is the maximum subarray of n problems of B, say $[a_0], [a_0, a_1], \dots, [a_0, \dots, a_{n-1}]$. The rule of reduction can happen into B, that is case $j = n - 1$ in the original problem is enough to construct it. We can write $p_n = \max(p_{n-1}, p_{n-1} + a_n, a_n)$.

In the array, a *suffix* is defined as any subarray which includes its last item. Another way to put the induction hypothesis into the problem B: We know how to find the maximum suffix of size $k < n$, we can easily induce the maximum suffix of size $k + 1$.

Another more creative option we convert this problem to best time to buy and sell stock problem. $[0, -2, -1, -4, 0, -1, 1, 2, -3, 1] \Rightarrow O(n)$, then we use prefix_sum, the difference is we set prefix_sum to 0 when it is smaller than 0, $O(n)$

```

1 from    sys import maxint
2 class Solution(object):
3     def maxSubArray(self, nums):
4         """
5             :type nums: List[int]
6             :rtype: int
7         """
8         max_so_far = -maxint - 1
9         prefix_sum= 0
10        for i in range(0, len(nums)):
11            prefix_sum+= nums[i]
12            if (max_so_far < prefix_sum):
13                max_so_far = prefix_sum
14
15            if prefix_sum< 0:
16                prefix_sum= 0
17        return max_so_far

```

13.6 The Skyline Problem

Define and solve it by two cases.

Both skyline problem and maximum subarray problem has illustrated how we can use reduction to solve our problem, either self-reduction or the A to B is used. The real algorithm design is usually a composite of multiple design steps and methods.

13.7 Exercises

1. Binary Search.

2. Use Self-Reduction by constant size to solve maximum subarray problem.
3. Skyline problem.

14

Decrease and Conquer

Want to do even better than linear complexity? Decrease and conquer reduces one problem into one smaller subproblem only, and the most common case is to reduce the state space into half of its original size. If the combining step takes only constant time, we get an elegant recurrence relation as:

$$T(n) = T(n/2) + O(1), \quad (14.1)$$

which gives us logarithmic time complexity!

We introduce three classical algorithms—binary search in array, binary search tree, and segment tree to enforce our understanding of decrease and conquer. Importantly, binary search and binary search tree consists **10%** of the total interview questions.

14.1 Introduction

All the searching we have discussed before never assumed any ordering between the items, and searching an item in an unordered space is doomed to have a time complexity linear to the space size. This case is about to change in this chapter.

Think about these two questions: What if we have a sorted list instead of an arbitrary one? What if the parent and children nodes within a tree are ordered in some way? With such special ordering between items in a data structures, can we increase its searching efficiency and be better than the blind one by one search in the state space? The answer is YES.

Let's take advantage of the ordering and the decrease and conquer methodology. To find a target in a space of size n , we first divide it into two subspaces and each of size $n/2$, say from the middle of the array. If the array is

increasingly ordered, all items in the left subspace are smaller than all items in the right subspace. If we compare our target with the item in the middle, we will know if this target is on the left or right side. With just one step, we reduced our state space by half size. We further repeat this process on the reduced space until we find the target. This process is called **Binary Search**. Binary search has recurrence relation:

$$T(n) = T(n/2) + O(1), \quad (14.2)$$

which decreases the time complexity from $O(n)$ to $O(\log n)$.

14.2 Binary Search

Binary search can be easily applied in sorted array or string.

```
For example, given a sorted and distinct array
nums = [1, 3, 4, 6, 7, 8, 10, 13, 14, 18, 19, 21, 24, 37, 40,
        45, 71]
Find target t = 7.
```

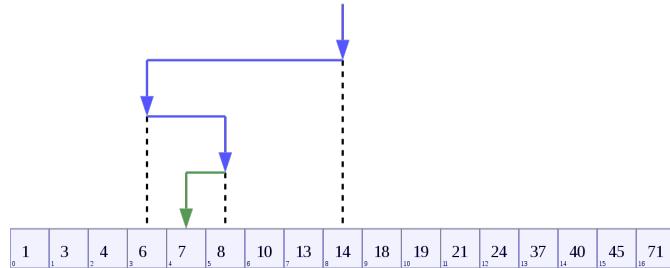


Figure 14.1: Example of Binary Search

Find the Exact Target This is the most basic application of binary search. We can set two pointers, l and r , which points to the first and last position, respectively. Each time we compute the middle position $m = (l+r)//2$, and check if the item $num[m]$ is equal to the target t .

- If it equals, target found and return the position.
- If it is smaller than the target, move to the left half by setting the right pointer to the position right before the middle position, $r = m - 1$.
- If it is larger than the target, move to the right half by setting the left pointer to the position right after the middle position, $l = m + 1$.

Repeat the process until we find the target or we have searched the whole space. The criterion of finishing the whole space is when l starts to be larger than r . Therefore, in the implementation we use a `while` loop with condition $l \leq r$ to make sure we only scan once of the searching space. The process of applying binary search on our exemplary array is depicted in Fig. 14.1 and the Python code is given as:

```

1 def standard_binary_search(lst, target):
2     l, r = 0, len(lst) - 1
3     while l <= r:
4         mid = l + (r - 1) // 2
5         if lst[mid] == target:
6             return mid
7         elif lst[mid] < target:
8             l = mid + 1
9         else:
10            r = mid - 1
11    return -1 # target is not found

```

In the code, we compute the middle position with `mid = l + (r - 1) // 2` instead of just `mid = (l + r) // 2` because these two always give the same computational result but the later is more likely to lead to overflow with its addition operator.

14.2.1 Lower Bound and Upper Bound

Duplicates and Target Missing What if there are duplicates in the array:

```

For example,
nums = [1, 3, 4, 4, 4, 4, 6, 7, 8]
Find target t = 4

```

Applying the first standard binary search will return 3 as the target position, which is the second 4 in the array. This does not seem like a problem at first. However, what if you want to know the predecessor or successor (3 or 5) of this target? In a distinct array, the predecessor and successor would be adjacent to the target. However, when the target has duplicates, the predecessor is before the first target and the successor is next to the last target. Therefore, returning an arbitrary one will not be helpful.

Another case, what if our target is 6, and we first want to see if it exists in the array. If it does not, we would like to insert it into the array and still keep the array sorted. The above implementation simply returns `-1`, which is not helpful at all.

The **lower and upper bound** of a binary search are the lowest and highest position where the value could be inserted without breaking the ordering.

v	1	3	4	4	4	4	6	7	8
i	0	1	2	3	4	5	6	7	8

$\nearrow >=4$

$\nwarrow <4$

I

Figure 14.2: Binary Search: Lower Bound of target 4.

v	1	3	4	4	4	4	6	7	8
i	0	1	2	3	4	5	6	7	8

$\nearrow <=4$

$\nwarrow >4$

u

Figure 14.3: Binary Search: Upper Bound of target 4.

For example, if our $t = 4$, the first position it can insert is at index 2 and the last position is at index 6.

- With index 2 as the lower bound, items in $i \in [0, l-1]$, $a[i] < t$, $a[l] = t$, and $i \in [l, n)$, $a[i] \geq t$. A lower bound is also the first position that has a value $v \geq t$. This case is shown in Fig. 14.2.
- With the upper bound, items in $i \in [0, u-1]$, $a[i] \leq t$, $a[u] = t$, and $i \in [u, n)$, $a[i] > t$. An upper bound is also the first position that has a value $v > t$. This case is shown in Fig. 14.3.

v	1	3	4	4	4	4	6	7	8
i	0	1	2	3	4	5	6	7	8

$\nearrow <5$

$\nwarrow >5$

I, u

Figure 14.4: Binary Search: Lower and Upper Bound of target 5 is the same.

If $t = 5$, the only position it can insert is at index 6, which indicates $l = u$. We show this case in Fig. 14.4.

Now that we know the meaning of the upper and lower bound, here comes to the question, “How to implement them?”

Implement Lower Bound Because if the target equals to the value at the middle index, we have to move to the left half to find its leftmost position of the same value. Therefore, the logic is that we move as left as possible until it can't further. When it stops, $l > r$, and 1 points to the first position that the value v be $v \geq t$. Another way to think about the return value is with assumption: Assume the middle pointer m is at the first position that equals to the target in the case of target 4, which is index 2. According to the searching rule, it goes to the left search space and changes the right pointer as $r = m - 1$. At this point, in the valid search space, there will never be a value that can be larger or equals to the target, pointing out that it will only moving to the right side, increasing the 1 pointer and leave the r pointer untouched until $l > r$ and the search stops. When the first time that $l > r$, the left pointer will be $l = r + 1 = m$, which is the first position that its value equals to the target.

The search process for target 4 and 5 is described as follows:

```
0: l = 0, r = 8, mid = 4
1: mid = 4, 4==4, l = 0, r = 3
2: mid = 1, 4>3, l = 2, r = 3
3: mid = 2, 4==4, l = 2, r = 1
return l=2
```

Similarly, we run the case for target 5.

```
0: l = 0, r = 8, mid = 4
1: mid = 4, 5>4, l = 5, r = 8
2: mid = 6, 5<6, l = 5, r = 5
3: mid = 5, 5>4, l = 6, r = 5
return l=6
```

The Python code is as follows:

```
1 def lower_bound_bs(nums, t):
2     l, r = 0, len(nums) - 1
3     while l <= r:
4         mid = l + (r - 1) // 2
5         if t <= nums[mid]: # move as left as possible
6             r = mid - 1
7         else:
8             l = mid + 1
9     return l
```

Implement Upper Bound To be able to find the upper bound, we need to move the left pointer to the right as much as possible. Assume we have the middle index at 5, with target as 4. The binary search moves to the

right side of the state space, making $l = mid + 1 = 6$. Now, in the right state space, the middle pointer will always have values larger than 4, thus it will only move to the left side of the space, which only changes the right pointer r and leaves the left pointer l untouched when the program ends. Therefore, l will still return our final upper bound index. The Python code is as follows:

```

1 def upper_bound_bs(nums, t):
2     l, r = 0, len(nums) - 1
3     while l <= r:
4         mid = l + (r - l) // 2
5         if t >= nums[mid]: # move as right as possible
6             l = mid + 1
7         else:
8             r = mid - 1
9     return l

```

Python Module `bisect` Conveniently, we have a Python built-in Module `bisect` that offers two methods: `bisect_left()` for obtaining the lower bound and `bisect_right()` to obtain the upper bound. For example, we can use it as:

```

1 from bisect import bisect_left, bisect_right, bisect
2 l1 = bisect_left(nums, 4)
3 r1 = bisect_right(nums, 5)
4 l2 = bisect_right(nums, 4)
5 r2 = bisect_right(nums, 5)

```

It offers six methods as shown in Table 14.1.

Table 14.1: Methods of `bisect`

Method	Description
<code>bisect_left(a, x, lo=0, hi=len(a))</code>	The parameters <code>lo</code> and <code>hi</code> may be used to specify a subset of the list; the function is the same as <code>bisect_left_raw</code>
<code>bisect_right(a, x, lo=0, hi=len(a))</code>	The parameters <code>lo</code> and <code>hi</code> may be used to specify a subset of the list; the function is the same as <code>bisect_right_raw</code>
<code>bisect(a, x, lo=0, hi=len(a))</code>	Similar to <code>bisect_left()</code> , but returns an insertion point which comes after (to the right of) any existing entries of <code>x</code> in <code>a</code> .
<code>insort_left(a, x, lo=0, hi=len(a))</code>	This is equivalent to <code>a.insert(bisect.bisect_left(a, x, lo, hi), x)</code> .
<code>insort_right(a, x, lo=0, hi=len(a))</code>	This is equivalent to <code>a.insert(bisect.bisect_right(a, x, lo, hi), x)</code> .
<code>insort(a, x, lo=0, hi=len(a))</code>	Similar to <code>insort_left()</code> , but inserting <code>x</code> in <code>a</code> after any existing entries of <code>x</code> .

Bonus For the lower bound, if we return the position as `l-1`, then we get the last position that `value < target`. Similarly, for the upper bound, we

get the last position `value <= target`.

14.2.2 Applications

Binary Search is a powerful problem solving tool. Let's go beyond the sorted array: How about when the array is sorted in a way that is not as monotonic as what we are familiar with, or how about solving math functions with binary search, whether they are continuous or discrete, equations or inequations?

First Bad Version(L278) You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions $[1, 2, \dots, n]$ and you want to find out the first bad one, which causes all the following ones to be bad. You are given an API `bool isBadVersion(version)` which will return whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Given $n = 5$, and $\text{version} = 4$ is the first bad version.

```
call isBadVersion(3) -> false
call isBadVersion(5) -> true
call isBadVersion(4) -> true
```

Then 4 is the first bad version.

Analysis and Design In this case, we have a search space in range $[1, n]$. Think the value at each position is the result from function `isBadVersion(i)`. Assume the first bad version is at position b , then the values from the positions are of such pattern: $[F, \dots, F, \dots, F, T, \dots, T]$. We can totally apply the binary search in the search space $[1, n]$: to find the first bad version is the same as finding the first position that we can insert a value `True`—the lower bound of value `True`. Therefore, whenever the value we find is `True`, we move to the left space to try to get its first location. The Python code is given below:

```
1 def firstBadVersion(n):
2     l, r = 1, n
3     while l <= r:
4         mid = l + (r - 1) // 2
5         if isBadVersion(mid):
6             r = mid - 1
7         else:
8             l = mid + 1
9     return l
```

Search in Rotated Sorted Array

“How about we rotate the sorted array?”

Problem Definition(L33, medium) Suppose an array (without duplicates) is first sorted in ascending order, but later is rotated at some pivot unknown to you beforehand—it takes all items before the pivot to the end of the array. For example, an array [0, 1, 2, 4, 5, 6, 7] be rotated at pivot 4, will become [4, 5, 6, 7, 0, 1, 2]. If the pivot is at 0, nothing will be changed. If it is at the end of the array, say 7, it becomes [7, 0, 1, 2, 4, 5, 6]. You are given a target value to search. If found in the array return its index, otherwise return -1.

Example 1:

```
Input: nums = [3,4,5,6,7,0,1,2], target = 0
Output: 5
```

```
target = 8
Output: -1
```

Analysis and Design In the rotated sorted array, the array is not purely monotonic. Instead, there will be at most one drop in the array because of the rotation, which we denote the high and the low item as a_h, a_l respectively. This drop cuts the array into two parts: $a[0 : h + 1]$ and $a[l : n]$, and both parts are ascending sorted. If the middle point falls within the left part, the left side of the state space will be sorted, and if it falls within the right part, the right side of the state space will be sorted. Therefore, at any situation, there will always be one side of the state space that is sorted. To check which side is sorted, simply compare the value of middle pointer with that of left pointer.

- If `nums[1] < nums[mid]`, then the left part is sorted.
- If `nums[1] > nums[mid]`, then the right part is sorted.
- Otherwise when they equal to each other, which is only possible that there is no left part left, we have to move to the right part. For example, when `nums=[1, 3]`, we move to the right part.

With a sorted half of state space, we can check if our target is within the sorted half: if it is, we switch the state space to the sorted space; otherwise, we have to move to the other half that is unknown. The Python code is shown as:

```
1 def RotatedBinarySearch(nums, t):
2     l, r = 0, len(nums)-1
3     while l <= r:
4         mid = l + (r-l)//2
```

```

5     if nums[mid] == t:
6         return mid
7 # Left is sorted
8     if nums[1] < nums[mid]:
9         if nums[1] <= t < nums[mid]:
10            r = mid - 1
11        else:
12            l = mid + 1
13 # Right is sorted
14     elif nums[1] > nums[mid]:
15         if nums[mid] < t <= nums[r]:
16             l = mid + 1
17         else:
18             r = mid - 1
19 # Left and middle index is the same, move to the right
20     else:
21         l = mid + 1
22
23 return -1

```



What happens if there are duplicates in the rotated sorted array?

In fact, the same comparison rule applies, with one minor change. When `nums=[1, 3, 1, 1, 1]`, the middle pointer and the left pointer has the same value, and in this case, the right side will only consist of a single value, making us to move to the left side instead. However, if `nums=[1, 1, 3]`, we need to move to the right side instead. Moreover, for `nums=[1, 3]`, it is because there is no left side we have to search the the right side. Therefore, in this case, it is impossible for us to decide which way to go, a simple strategy is to just move the left pointer forward by one position and retreat to the linear search.

```

1 # The left half is sorted
2 if nums[mid] > nums[1]:
3 # The right half is sorted
4 elif nums[mid] < nums[1]:
5 # For third case
6 else:
7     l +=1

```

Binary Search to Solve Functions

Now, let's see how it can be applied to solve equations or inequations. Assume, our function is $y = f(x)$, and this function is monotonic, such as $y = x$, $y = x^2 + 1$, $y = \sqrt{x}$. To solve this function is the same as finding a solution x_t to a given target y_t . We generally have three steps to solve such problems:

1. Set a search space for x , say it is $[x_l, x_r]$.

2. If the function is equation, we find a x_t that either equals to y_t or close enough such as $|y_t - y| \leq 1e - 6$ using standard binary search.
3. If the function is inequation, we see if it wants the first or the last x_t that satisfy the constraints on y . It is the same as of finding the lower bound or upper bound.

Arranging Coins (L441, easy) You have a total of n coins that you want to form in a staircase shape, where every k -th row must have exactly k coins. Given n , find the total number of full staircase rows that can be formed. n is a non-negative integer and fits within the range of a 32-bit signed integer.

Example 1:

```
n = 5
```

The coins can form the following rows:

```
*
```

```
* *
```

```
* *
```

Because the 3rd row is incomplete, we return 2.

Analysis and Design Each row x has x coins, summing it up, we get $1 + 2 + \dots + x = \frac{x(x+1)}{2}$. The problem is equivalent to find the last integer x that makes $\frac{x(x+1)}{2} \leq n$. Of course, this is just a quadratic equation which can be easily solved if you remember the formula, such as the following Python code:

```
1 import math
2 def arrangeCoins(n: int) -> int:
3     return int((math.sqrt(1+8*n)-1) // 2)
```

However, if in the case where we do not know a direct closed-form solution, we solicit binary search. First, the function of x is monotonically increasing, which indicates that binary search applies. We set the range of x to $[1, n]$, what we need is to find the last position that the condition of $\frac{x(x+1)}{2} \leq n$ satisfies, which is the position right before the upper bound. The Python code is given as:

```
1 def arrangeCoins(n):
2     def isValid(row):
3         return (row * (row + 1)) // 2 <= n
4
5     def bisect_right():
6         l, r = 1, n
7         while l <= r:
8             mid = l + (r-1) // 2
9             # Move as right as possible
10            if isValid(mid):
11                l = mid + 1
```

```

12     else :
13         r = mid - 1
14     return 1
15 return bisect_right() - 1

```

14.3 Binary Search Tree

A sorted array supports logarithmic query time with binary search, however it still takes linear time to update–delete or insert items. Binary search tree (BSTs), a type of binary tree designed for fast access and updates to items, on the other hand, only takes $O(\log n)$ time to update. How does it work?

In the array data structure, we simply sort the items, but how to apply sorting in a binary tree? Review the min-heap data structure, which recursively defining a node to have the largest value among the nodes that belong to the subtree of that node, will give us a clue. In the binary search tree, we define that for any given node x , all nodes in the left subtree of x have keys smaller than x while all nodes in the right subtree of x have keys larger than x . An example is shown in Fig. 27.1. With this definition, simply comparing a search target with the root can point us to half of the search space, given the tree is balanced enough. Moreover, if we do in-order traversal of nodes in the tree from the root, we end up with a nice and sorted keys in ascending order, making binary search tree one member of the sorting algorithms.

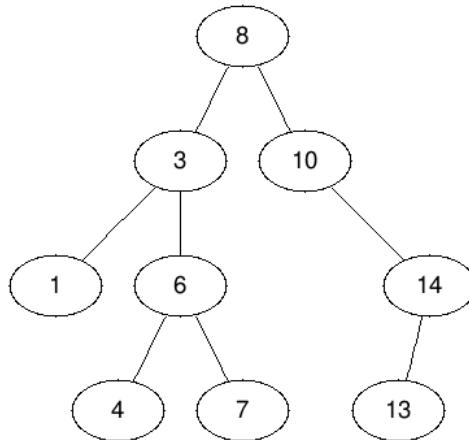


Figure 14.5: Example of Binary search tree of depth 3 and 8 nodes.

Binary search tree needs to support many operations, including searching for a given key, the minimum and maximum key, and a predecessor or successor of a given key, inserting and deleting items while maintaining the binary search tree property. Because of its efficiency of these operations compared with other data structures, binary search tree is often used as a dictionary or a priority queue.

With l and r to represent the left and right child of node x , there are two other definitions other than the binary search tree definition we just introduced: (1) $l.key \leq x.key < r.key$ and (2) $l.key < x.key \leq r.key$. In these two cases, our resulting BSTs allows us to have duplicates. The exemplary implementation follow the definition that does not allow duplicates.

14.3.1 Operations

In order to build a BST, we need to insert a series of items in the tree organized by the search tree property. And in order to insert, we need to search for a proper position first and then insert the new item while sustaining the search tree property. Thus, we introduce these operations in the order of search, insert and generate.

Search The search is highly similar to the binary search in the array. It starts from the root. Unless the node's value equals to the target, the search proceeds to either the left or right child depending upon the comparison result. The search process terminates when either the target is found or when an empty node is reached. It can be implemented either recursively or iteratively with a time complexity $O(h)$, where h is the height of the tree, which is roughly $\log n$ if the tree is balanced enough. The recursive search is shown as:

```

1 def search(root, t):
2     if not root:
3         return None
4     if root.val == t:
5         return root
6     elif t < root.val:
7         return search(root.left, t)
8     else:
9         return search(root.right, t)

```

Because this is a tail recursion, it can easily be converted to iteration, which helps us save the heap space. The iterative code is given as:

```

1 # iterative searching
2 def iterative_search(root, key):
3     while root is not None and root.val != key:
4         if root.val < key:
5             root = root.right
6         else:
7             root = root.left
8     return root

```



Write code to find the minimum and maximum key in the BST.

The minimum key locates at the leftmost of the BST, while the maximum key locates at the rightmost of the tree.

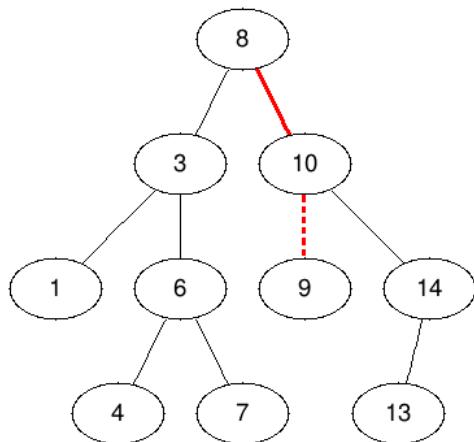


Figure 14.6: The red colored path from the root down to the position where the key 9 is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Insert Assuming we are inserting a node with key 9 into the tree shown in Fig 27.1. We start from the root, compare 9 with 8, and goes to node 10. Next, the search process will lead us to the left child of node 10, and this is where we should put node 9. The process is shown in Fig. 14.6.

The process itself is easy and clean. Here comes to the implementation. We treat each node as a subtree: whenever the search goes into that node, then the algorithm hands over the insertion task totally to that node, and assume it has inserted the new node and return its updated node. The main program will just simply reset its left or right child with the return value from its children. The insertion of new node happens when the search hits an empty node, it returns a new node with the target value. The implementation is given as:

```

1 def insert(root, t):
2     if not root:
3         return BiNode(t)
4     if root.val == t:
5         return root
6     elif t < root.val:
7         root.left = insert(root.left, t)
8         return root
  
```

```

9     else :
10        root.right = insert(root.right , t)
11    return root

```

In the notebook, I offered a variant of implementation, check it out if you are interested. To insert iteratively, we need to track the parent node while searching. The `while` loop stops when it hit at an empty node. There will be three cases in the case of the parent node:

1. When the parent node is `None`, which means the tree is empty. We assign the root node with the a new node of the target value.
2. When the target's value is larger than the parent node's, the put a new node as the right child of the parent node.
3. When the target's value is smaller than the parent node's, the put a new node as the left child of the parent node.

The iterative code is given as:

```

1 def insertItr(root , t):
2     p = None
3     node = root #Keep the root node
4     while node:
5         # Node exists already
6         if node.val == t:
7             return root
8         if t > node.val:
9             p = node
10            node = node.right
11        else:
12            p = node
13            node = node.left
14    # Assign new node
15    if not p:
16        root = BiNode(t)
17    elif t > p.val:
18        p.right = BiNode(t)
19    else:
20        p.left = BiNode(t)
21    return root

```

BST Generation To generate our exemplary BST shown in Fig. 27.1, we set `keys = [8, 3, 10, 1, 6, 14, 4, 7, 13]`, then we call `insert` function we implemented for each key to generate the same tree. The time complexity will be $O(n \log n)$.

```

1 keys = [8 , 3 , 10 , 1 , 6 , 14 , 4 , 7 , 13]
2 root = None
3 for k in keys:
4     root = insert(root , k)

```

Find the Minimum and Maximum Key Because the minimum key is the leftmost node within the tree, the search process will always traverse to the left subtree and return the last non-empty node, which is our minimum node. The time complexity is the same as of searching any key, which is $O(\log n)$.

```

1 def minimum(root):
2     if not root:
3         return None
4     if not root.left:
5         return root
6     return minimum(root.left)

```

It can easily be converted to iterative:

```

1 def minimumIter(root):
2     while root:
3         if not root.left:
4             return root
5         root = root.left
6     return None

```

To find the maximum node, replacing `left` with `right` will do. Also, sometimes we need to search two additional items related to a given node: successor and predecessor. The structure of a binary search tree allows us to determine the successor or the predecessor of a tree without ever comparing keys.

Successor A successor of node x is the smallest node in BST that is strictly greater than x . It is also called **in-order successor**, which is the node next to node x in the inorder traversal ordering—sorted ordering. Other than the maximum node in BST, all other nodes will have a successor. The simplest implementation is to return the next node within inorder traversal. This will have a linear time complexity, which is not great. The code is shown as:

```

1 def successorInorder(root, node):
2     if not node:
3         return None
4     if node.right is not None:
5         return minimum(node.right)
6     # Inorder traversal
7     succ = None
8     while root:
9         if node.val > root.val:
10            root = root.right
11        elif node.val < root.val:
12            succ = root
13            root = root.left
14        else:
15            break
16    return succ

```

Let us try something else. In the BST shown in Fig. 14.6, the node 3's successor will be node 4. For node 4, its successor will be node 6. For node 7, its successor is node 8. What are the cases here?

- An easy case is when a node has right subtree, its successor is the minimum node within its right subtree.
- However, if a node does not have a right subtree, there are two more cases:
 - If it is a left child of its parent, such as node 4 and 9, its direct parent is its successor.
 - However, if it is a right child of its parent, such as node 7 and 14, we traverse backwards to check its parents. If a parent node is the left child of its parent, then that parent will be the successor. For example, for node 7, we traverse through 6, 3, and 3 is a left child of node 8, making node 8 the successor for node 7.

The above two rules can be merged as: starting from the target node, traverse backward to check its parent, find the first two nodes which are in left child-parent relation. The parent node in that relation will be our targeting successor. Because the left subtree is always smaller than a node, when we backward, if a node is smaller than its parent, it tells us that the current node is smaller than that parent node too.

We write three functions to implement the successor:

- Function `findNodeAddParent` will find the target node and add a `parent` node to each node along the searching that points to their parents. The Code is as:

```

1 def findNodeAddParent(root , t):
2     if not root:
3         return None
4     if t == root.val:
5         return root
6     elif t < root.val:
7         root.left.p = root
8         return findNodeAddParent(root.left , t)
9     else:
10        root.right.p = root
11        return findNodeAddParent(root.right , t)

```

- Function `reverse` will find the first left-parent relation when traverse backward from a node to its parent.

```

1 def reverse(node):
2     if not node or not node.p:
3         return None
4     # node is a left child

```

```

5     if node.val < node.p.val:
6         return node.p
7     return reverse(node.p)

```

- Function `successor` takes a node as input, and return its successor.

```

1 def successor(root):
2     if not root:
3         return None
4     if root.right:
5         return minimum(root.right)
6     else:
7         return reverse(root)

```

To find a successor for a given key, we use the following code:

```

1 root.p = None
2 node = findNodeAddParent(root, 4)
3 suc = successor(node)

```

This approach will give us $O(\log n)$ time complexity.

Predecessor A predecessor of node x on the other side, is the largest item in BST that is strictly smaller than x . It is also called **in-order predecessor**, which denotes the previous node in Inorder traversal of BST. For example, for node 6, the predecessor is node 4, which is the maximum node within its left subtree. For node 4, its predecessor is node 3, which is the parent node in a right child-parent relation while tracing back through parents. Now, assume we find the targeting node with function `findNodeAddParent`, we first write `reverse` function as `reverse_right`.

```

1 def reverse_right(node):
2     if not node or not node.p:
3         return None
4     # node is a right child
5     if node.val > node.p.val:
6         return node.p
7     return reverse_right(node.p)

```

Next, we implement the above rules to find predecessor of a given node.

```

1 def predecessor(root):
2     if not root:
3         return None
4     if root.left:
5         return maximum(root.left)
6     else:
7         return reverse_right(root)

```

The expected time complexity is $O(\log n)$. And the worst is when the tree line up and has no branch, which makes it $O(n)$. Similarly, we can use inorder traversal:

```

1 def predecessorInorder(root, node):
2     if not node:
3         return None
4     if node.left is not None:
5         return maximum(node.left)
6     # Inorder traversal
7     pred = None
8     while root:
9         if node.val > root.val:
10            pred = root
11            root = root.right
12        elif node.val < root.val:
13            root = root.left
14        else:
15            break
16    return pred

```

Delete When we delete a node, we need to restructure the subtree of that node to make sure the BST property is maintained. There are different cases:

1. Node to be deleted is leaf: Simply remove from the tree. For example, node 1, 4, 7, and 13.
2. Node to be deleted has only one child: Copy the child to the node and delete the child. For example, to delete node 14, we need to copy node 13 to node 14.
3. Node to be deleted has two children, for example, to delete node 3, we have its left and right subtree. We need to get a value, which can either be its predecessor-node 1 or successor-node 4, and copy that value to the position about to be deleted.

To support the delete operation, we write a function `deleteMinimum` to obtain the minimum node in that subtree and return a subtree that has that node deleted.

```

1 def deleteMinimum(root):
2     if not root:
3         return None, None
4     if root.left:
5         mini, left = deleteMinimum(root.left)
6         root.left = left
7         return mini, root
8     # the minimum node
9     if not root.left:
10        return root, None

```

Next, we implement the above three cases in function `_delete` when a deleting node is given, which will return a processed subtree deleting its root node.

```

1 def __delete(root):
2     if not root:
3         return None
4     # No children: Delete it
5     if not root.left and not root.right:
6         return None
7     # Two children: Copy the value of successor
8     elif all([root.left, root.right]):
9         succ, right = deleteMinimum(root.right)
10        root.val = succ.val
11        root.right = right
12        return root
13    # One Child: Copy the value
14    else:
15        if root.left:
16            root.val = root.left.val
17            root.left = None
18        else:
19            root.val = root.right.val
20            root.right = None
21    return root

```

Finally, we call the above two function to delete a node with a target key.

```

1 def delete(root, t):
2     if not root:
3         return
4     if root.val == t:
5         root = __delete(root)
6         return root
7     elif t > root.val:
8         root.right = delete(root.right, t)
9         return root
10    else:
11        root.left = delete(root.left, t)
12    return root

```

14.3.2 Binary Search Tree with Duplicates

If we use any of the other two definitions we introduced that allows duplicates, things can be more complicated. For example, if we use the definition $x.left.key \leq x.key < x.right.key$, we will end up with a tree looks like Fig. 14.7:

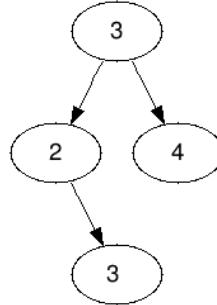


Figure 14.7: A BST with nodes 3 duplicated twice.

Note that the duplicates are not in contiguous levels. This is a big issue when allowing duplicates in a BST representation as, because duplicates may be separated by any number of levels, making the detection of duplicates difficult.

An option to avoid this issue is to not represent duplicates structurally (as separate nodes) but instead use a **counter** that counts the number of occurrences of the key. The previous example will be represented as in Fig. 14.8:

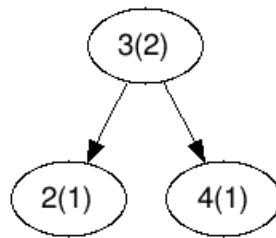


Figure 14.8: A BST with nodes 3 marked with two occurrence.

This simplifies the related operations at the expense of some extra bytes and counter operations. Since a heap is a complete binary tree, it has a smallest possible height - a heap with N nodes always has $O(\log N)$ height.

14.4 Segment Tree

To answer queries over an array is called a *range query problem*, e.g. finding the sum of consecutive subarray $a[l : r]$, or finding the minimum item in such a range. A direct and linear solution is to compute the required query on the subarray on the fly each time. When the array is large, and the update is frequent, even this linear approach will be too slow. Let's try to solve this problem faster than linear. How about computing the query for a range in advance and save it in a dictionary? If we can, the query time is constant.

However, because there are n^2 subarray, making the space cost polynomial, which is definitely not good. Another problem, “what if we need to change the value of an item”, we have to update n nodes in the dictionary which includes the node in its range.

We can balance the search, update, and space from the dictionary approach to a logarithmic time with the technique of decrease and conquer. In the binary search, we keep dividing our search space into halves recursively until a search space can no longer be divided. We can apply the dividing process here, and construct a binary tree, and each node has l and r to indicate the range of that node represents. For example, if our array has index range $[0, 5]$, its left subtree will be $[0, \text{mid}]$, and right subtree will be $[\text{mid}+1, 5]$. A binary tree built with binary search manner is shown in Fig. 14.9.

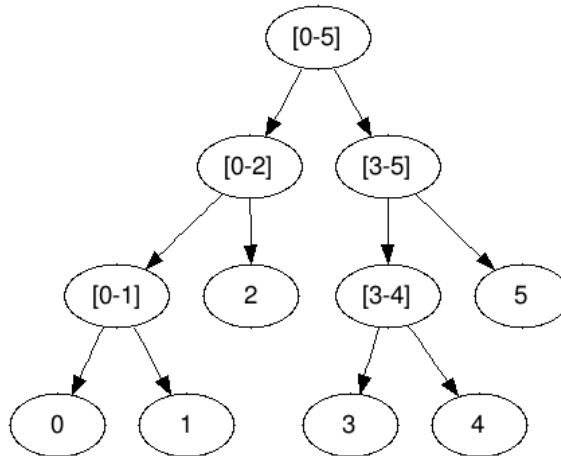


Figure 14.9: A Segment Tree

To get the answer for range query $[0, 5]$, we just return the value at root node. If the range is $[0, 1]$, which is on the left side of the tree, we go to the left branch, and cutting half of the search space. For a range that happens to be between two nodes, such as $[1, 3]$, which needs node $[0, 1]$ and $[2-5]$, we search $[0, 1]$ in the left subtree and $[2, 3]$ in the right subtree and combine them together. Any searching will be within $O(\log n)$, relating to the height of the tree. **needs better complexity analysis**

Segment tree The above binary tree is called **segment tree**. From our analysis, we can see a segment tree is a static full binary trees. 'Static' here means once the data structure is built, it can not be modified or extended. However, it can still update the value in the original array into the segment tree. Segment tree is applied widely to efficiently answer numerous *dynamic range queries* problems (in logarithmic time), such as finding minimum,

maximum, sum, greatest common divisor, and least common denominator in array.

Consider an array A of size n and a corresponding segment tree T :

1. The root of T represents the whole array $A[0 : n]$.
2. Each internal node in T represents the interval of $A[i : j]$ where $0 < i < j \leq n$.
3. Each leaf in T represents a single element $A[i]$, where $0 \leq i < n$.
4. If the parent node is in range $[i, j]$, then we separate this range at the middle position $m = (i + j)//2$; the left child takes range $[i, m]$, and the right child take the interval of $[m + 1, j]$.

Because in each step of building the segment tree, the interval is divided into two halves, so the height of the segment tree will be $\log n$. And there will be totally n leaves and $n - 1$ number of internal nodes, which makes the total number of nodes in segment tree to be $2n - 1$, which indicates a linear space cost. Except of an explicit tree can be used to implement segment tree, an implicit tree implemented with array can be used too, similar to the case of heap data structure.

14.4.1 Implementation

Implementation of a functional segment tree consists of three core operations: tree construction, range query, and value update, named as as `_buildSegmentTree()`, `RangeQuery()`, and `update()`, respectively. We demonstrate the implementation with Range Sum Query (RSQ) problem, but we try to generalize the process so that the template can be easily reused to other range query problems. In our implementation, we use explicit tree data structure for both convenience and easier to understand. We define a general tree node data structure as:

```

1 class TreeNode:
2     def __init__(self, val, s, e):
3         self.val = val
4         self.s = s
5         self.e = e
6         self.left = None
7         self.right = None

```

Range Sum Query(L307, medium) Given an integer array, find the sum of the elements between indices i and j , range $[i, j], i \leq j$.

Example :

Given $\text{nums} = [2, 9, 4, 5, 8, 7]$

```
sumRange(0, 2) -> 15
update(1, 3)
sumRange(0, 2) -> 9
```

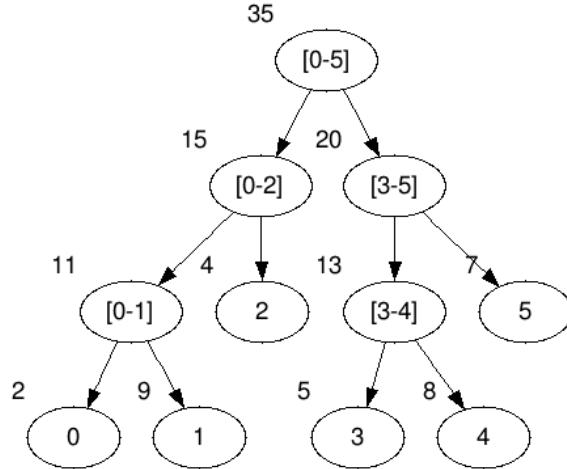


Figure 14.10: Illustration of Segment Tree for Sum Range Query.

Tree Construction The function `_buildSegmentTree()` takes three arguments: `nums`, `s` as the start index, and `e` as the end index. Because there are totally $2n - 1$ nodes, which makes the time and space complexity both be $O(n)$.

```

1 def __buildSegmentTree(nums, s, e):
2     """
3         s, e: start index and end index
4     """
5     if s > e:
6         return None
7     if s == e:
8         return TreeNode(nums[s], s, e)
9
10    m = (s + e)//2
11    # Divide: return a subtree
12    left = __buildSegmentTree(nums, s, m)
13    right = __buildSegmentTree(nums, m+1, e)
14
15    # Conquer: merge two subtree
16    node = TreeNode(left.val + right.val, s, e)
17    node.left = left
18    node.right = right
19    return node
  
```

Building a segment tree for our example as:

```
1 nums = [2, 9, 4, 5, 8, 7]
```

```
2 root = _buildSegmentTree(nums, 0, len(nums) - 1)
```

It will generate a tree shown in Fig. 14.10.

Range Query Each query within range $[i, j], i < j, i \geq s, j \leq e$, will be found on a node or by combining multiple node. In the query process, check the following cases:

- If range $[i, j]$ matches the range $[s, e]$, if it matches, return the value of the node, otherwise, processed to other cases.
- Compute middle index $m = (s + e)//2$. Check if range $[i, j]$ is within the left state space $[s, m]$ if $j \leq m$, or within the right state space $[m + 1, e]$ if $i \geq m + 1$, or is cross two spaces if otherwise.
 - For the first two cases, a recursive call on that branch will return our result.
 - For the third case, where the range crosses two space, two recursive calls on both children of our current node are needed: the left one handles range $[i, m]$, and the right one handles range $[m + 1, j]$. The final result will be a combination of these two.

The code is as follows:

```
1 def _rangeQuery(root, i, j, s, e):
2     if s == i and j == e:
3         return root.val if root else 0
4     m = (s + e)//2
5     if j <= m:
6         return _rangeQuery(root.left, i, j, s, m)
7     elif i > m:
8         return _rangeQuery(root.right, i, j, m+1, e)
9     else:
10        return _rangeQuery(root.left, i, m, s, m) + _rangeQuery(root
. right, m+1, j, m+1, e)
```

Update To update $\text{nums}[1]=3$, all nodes on the path from root to the leaf node will be affected and needed to be updated with to incorporate the change at the leaf node. We search through the tree with a range $[1, 1]$ just like we did within `_rangeQuery` except that we no longer need the case of crossing two ranges. Once we reach to the leaf node, we update that node's value to the new value, and it backtracks to its parents where we recompute the parent node's value according to the result of its children. This operation takes $O(\log n)$ time complexity, and we can do it inplace since the structure of the tree is not changed.

```
1 def _update(root, s, e, i, val):
2     if s == e == i:
3         root.val = val
```

```

4     return
5     m = (s + e) // 2
6     if i <= m:
7         _update(root.left, s, m, i, val)
8     else:
9         _update(root.right, m + 1, e, i, val)
10    root.val = root.left.val + root.right.val
11

```

Minimum and Maximum Range Query To get the minimum or maximum value within a given range, we just need to modify how to value is computed. For example, to update, we just need to change the line 10 of the above code to `root.val = min(root.left.val, root.right.val)`.

There are way more other variants of segment tree, check it out if you are into knowing more at https://cp-algorithms.com/data_structures/segment_tree.html.

14.5 Exercises

1. 144. Binary Tree Preorder Traversal
2. 94. Binary Tree Inorder Traversal
3. 145. Binary Tree Postorder Traversal
4. 589. N-ary Tree Preorder Traversal
5. 590. N-ary Tree Postorder Traversal
6. 429. N-ary Tree Level Order Traversal
7. 103. Binary Tree Zigzag Level Order Traversal(medium)
8. 105. Construct Binary Tree from Preorder and Inorder Traversal

938. Range Sum of BST (Medium)

Given the root node of a **binary search tree**, return the sum of values of all nodes with value between L and R (inclusive).

The binary search tree is guaranteed to have unique values.

```

1 Example 1:
2
3 Input: root = [10,5,15,3,7,null,18], L = 7, R = 15
4 Output: 32
5
6 Example 2:
7
8 Input: root = [10,5,15,3,7,13,18,1,null,6], L = 6, R = 10
9 Output: 23

```

Tree Traversal+Divide and Conquer. We need at most $O(n)$ time complexity. For each node, there are three cases: 1) $L \leq val \leq R$, 2) $val < L$, 3) $val > R$. For the first case it needs to obtain results for both its subtrees and merge with its own val. For the others two, because of the property of BST, only the result of one subtree is needed.

```

1 def rangeSumBST(self, root, L, R):
2     if not root:
3         return 0
4     if L <= root.val <= R:
5         return self.rangeSumBST(root.left, L, R) + self.
rangeSumBST(root.right, L, R) + root.val
6     elif root.val < L: #left is not needed
7         return self.rangeSumBST(root.right, L, R)
8     else: # right subtree is not needed
9         return self.rangeSumBST(root.left, L, R)

```

14.5.1 Exercises

14.1 35. Search Insert Position (easy). Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You can assume that there are no duplicates in the array.

Example 1:

Input: [1, 3, 5, 6], 5
Output: 2

Example 2:

Input: [1, 3, 5, 6], 2
Output: 1

Example 3:

Input: [1, 3, 5, 6], 7
Output: 4

Example 4:

Input: [1, 3, 5, 6], 0
Output: 0

Solution: Standard Binary Search Implementation. For this problem, we just standardize the Python code of binary search, which takes $O(\log n)$ time complexity and $O(1)$ space complexity without using recursion function. In the following code, we use exclusive right index with $\text{len}(\text{nums})$, therefore it stops if $l == r$; it can be as small as 0 or as large as n of the array length for numbers that are either smaller or equal to the $\text{nums}[0]$ or larger or equal to $\text{nums}[-1]$. We can also make the right index inclusive.

```

1 # exclusive version
2 def searchInsert(self, nums, target):
3     l, r = 0, len(nums) #start from 0, end to the len (
4         exclusive)
5     while l < r:
6         mid = (l+r)//2
7         if nums[mid] < target: #move to the right side
8             l = mid+1
9         elif nums[mid] > target: #move to the left side,
10            not mid-1
11             r= mid
12         else: #found the target
13             return mid
14     #where the position should go
15     return l

```

```

1 # inclusive version
2 def searchInsert(self, nums, target):
3     l = 0
4     r = len(nums)-1
5     while l <= r:
6         m = (l+r)//2
7         if target > nums[m]: #search the right half
8             l = m+1
9         elif target < nums[m]: # search for the left half
10            r = m-1
11        else:
12            return m
13    return l

```

Standard binary search

1. 611. Valid Triangle Number (medium)
2. 704. Binary Search (easy)
3. 74. Search a 2D Matrix) Write an efficient algorithm that searches for a value in an $m \times n$ matrix. This matrix has the following properties:
 - (a) Integers in each row are sorted from left to right.
 - (b) The first integer of each row is greater than the last integer of the previous row.

For example ,
Consider the following matrix:

```

[
    [1, 3, 5, 7],
    [10, 11, 16, 20],
    [23, 30, 34, 50]
]

```

Given target = 3, return true .

Also, we can treat *is* as one dimensional, and the time complexity is $O(\lg(m * n))$, which is the same as $O(\log(m) + \log(n))$.

```

1 class Solution:
2     def searchMatrix(self, matrix, target):
3         if not matrix or target is None:
4             return False
5
6         rows, cols = len(matrix), len(matrix[0])
7         low, high = 0, rows * cols - 1
8
9         while low <= high:
10            mid = (low + high) / 2
11            num = matrix[mid / cols][mid % cols]
12
13            if num == target:
14                return True
15            elif num < target:
16                low = mid + 1
17            else:
18                high = mid - 1
19
20        return False

```

Check <http://www.cnblogs.com/grandyang/p/6854825.html> to get more examples.

Search on rotated and 2d matrix:

1. 81. Search in Rotated Sorted Array II (medium)
2. 153. Find Minimum in Rotated Sorted Array (medium) The key here is to compare the mid with left side, if mid-1 has a larger value, then that is the minimum
3. 154. Find Minimum in Rotated Sorted Array II (hard)

Search on Result Space:

1. 367. Valid Perfect Square (easy) (standard search)
2. 363. Max Sum of Rectangle No Larger Than K (hard)
3. 354. Russian Doll Envelopes (hard)
4. 69. Sqrt(x) (easy)

15

Sorting and Selection Algorithms

Sorting is the most basic building block for many other algorithms and is often considered as the very first step that eases and reduces the original problems to easier ones.

15.1 Introduction

Sorting In computer science, a *sorting algorithm* is designed to rearrange items of a given array in a certain order based on each item's *key*. The most frequently used orders are *numerical order* and *lexicographical order*. For example, given an array of size n , sort items in increasing order of its numerical values:

```
Array = [9, 10, 2, 8, 9, 3, 7]
sorted = [2, 3, 7, 8, 9, 9, 10]
```

Selection *Selection algorithm* is used to find the k -th smallest number in a given array; such a number is called the k -th *order statistic*. For example, given the above array, find the 3-th smallest number.

```
Array = [9, 10, 2, 8, 9, 3, 7], k = 3
Result: 7
```

Sorting and Selection often go hand in hand; either we first execute sorting and then select the desired order through indexing or we derive a selection algorithm from a corresponding sorting algorithm. Due to such relation, this chapter is mainly about introducing sorting algorithms and occasionally we introduce their corresponding selection algorithms by the side.

Lexicographical Order For a list of strings, sorting them will make them in lexicographical order. The order is decided by a comparison function, which compares corresponding characters of the two strings from left to right. In the process, the first pair of characters that differ from each other determines the ordering: the string that has smaller alphabet from the pair is smaller than the other string.

Characters are compared using the Unicode character set. All uppercase letters come before lower case letters. If two letters are the same case, then alphabetic order is used to compare them. For example:

```
'ab' < 'bc' (differs at i = 0)
'abc' < 'abd' (differs at i = 2)
```

Special cases appears when two strings are of different length and the shorter one s is a prefix of the longer one t , then it is considered that $s < t$. For example:

```
'ab' < 'abab' ('ab' is a prefix of 'abab')
```

How to Learn Sorting Algorithms? We list a few terminologies that are commonly seen to describe the properties of a certain sorting algorithm:

- **In-place Sorting:** In-place sorting algorithm only uses a constant number of extra spaces to assist its implementation. If a sorting algorithm is not in-place, it is called out-of-place sorting instead.
- **Stable Sorting:** Stable sorting algorithm maintain the relative order of items with equal keys. For example, two different tasks that come with same priority in the priority queue should be scheduled in the relative pending ordering.
- **Comparison-based Sorting:** This kind of sorting technique determines the sorted order of an input array by comparing pairs of items and moving them around based on the results of comparison. And it has a lower bound of $\Omega(n \log n)$ comparison.

Sorting Algorithms in Coding Interviews As the fundamental Sorting and selection algorithms can still be potentially met in interviews where we might be asked to implement and analyze any sorting algorithm you like. Therefore, it is necessary for us to understand the most commonly known sorting algorithms. Also, Python provides us built-in sorting algorithms to use directly and we shall master the syntax too.

The Applications of Sorting The importance of sorting techniques is decided by its multiple fields of application:

1. Sorting can organize information in a human-friendly way. For example, the lexicographical order are used in dictionary and inside of library systems to help users locate wanted words or books in a quick way.
2. Sorting algorithms often be used as a key subroutine to other algorithms. As we have shown before, binary search, sliding window algorithms, or cyclic shifts of suffix array need the data to be in sorted order to carry on the next step. When ordering will not incur wrong solution to the problems, sorting beforehand should always be atop on our mind for sorting first might ease our problem later.

Organization We organize the content mainly based on the worst case time complexity. Section 15.3 - 15.4 focuses on comparison-based sorting algorithms, and Section 15.5.2-?? introduce classical non-comparison-based sorting algorithms.

- Naive Sorting (Section 15.3): Bubble Sort, Insertion Sort, Selection Sort;
- Asymptotically Best Sorting (Section 15.4) Sorting: merge sort, quick sort, and Quick Select;
- Linear Sorting (Section 15.5.2): Counting Sort, where k is the range of the very first and last key.
- Python Built-in Sort (Section 15.6):

15.2 Python Comparison Operators and Built-in Functions

Comparison Operators Python offers 7 comparison operators shown in Table. 30.2 to compare values. It either returns `True` or `False` according to the condition.

Table 15.1: Comparison operators in Python

>	Greater than - <code>True</code> if left operand is greater than the right
<	Less than - <code>True</code> if left operand is less than the right
==	Equal to - <code>True</code> if both operands are equal
!=	Not equal to - <code>True</code> if operands are not equal
>=	Greater than or equal to - <code>True</code> if left operand is greater than or equal to the right
<=	Less than or equal to - <code>True</code> if left operand is less than or equal to the right

For example, compare two numerical values:

```
1 c1 = 2 < 3
2 c2 = 2.5 > 3
```

The printout is:

```
1 (True, False)
```

Also, compare two strings follows the lexicographical orders:

```
1 c1 = 'ab' < 'bc'
2 c2 = 'abc' > 'abd'
3 c3 = 'ab' < 'abab'
4 c4 = 'abc' != 'abc'
```

The printout is:

```
1 (True, False, True, False)
```

What's more, it can compare other types of sequences such as `list` and `tuple` using lexicographical orders too:

```
1 c1 = [1, 2, 3] < [2, 3]
2 c2 = (1, 2) > (1, 2, 3)
3 c3 = [1, 2] == [1, 2]
```

The printout is:

```
1 (True, False, True)
```

However, mostly Python 3 does not support comparison between different types of sequence, nor does it supports comparison for `dictionary`. For `dictionary` data structures, in default, it uses its key as the key to compare with. For example, comparison between `list` and `tuple` will raise `TypeError`:

```
1 [1, 2, 3] < (2, 3)
```

The error is shown as:

```
1 -----> 1 [1, 2, 3] < (2, 3)
2 TypeError: '<' not supported between instances of 'list' and 'tuple'
```

Comparison between dictionary as follows will raise the same error:

```
1 {1: 'a', 2: 'b'} < {1: 'a', 2: 'b', 3: 'c'}
```

Comparison Functions Python built-in functions `max()` and `min()` support two forms of syntax: `max(iterable, *[, key, default])` and `max(arg1, arg2, *args[, key])`. If one positional argument is provided, it should be an iterable. And then it returns the largest item in the iterable based on its key. It also accepts two or more positional arguments, and these arguments can be numerical or sequential. When there are two or more positional argument, the function returns the largest.

For example, with one iterable and it returns 20:

```
1 max([4, 8, 9, 20, 3])
```

With two positional arguments –either numerical or sequential:

```
1 m1 = max(24, 15)
2 m2 = max([4, 8, 9, 20, 3], [6, 2, 8])
3 m3 = max('abc', 'ba')
```

The printout of these results is:

```
1 (24, [6, 2, 8], 'ba')
```

With **dictionary**:

```
1 dict1 = {'a': 5, 'b': 8, 'c': 3}
2 k1 = max(dict1)
3 k2 = max(dict1, key=dict1.get)
4 k3 = max(dict1, key=lambda x: dict1[x])
```

The printout is:

```
1 ('c', 'b', 'b')
```

When the sequence is empty, we need to set an default value:

```
1 max([], default=0)
```

Rich Comparison To compare a self-defined **class**, in Python 2.X, `__cmp__(self, other)` special method is used to implement comparison between two objects. `__cmp__(self, other)` returns negative value if `self < other`, positive if `self > other`, and zero if they were equal. However, in Python 3, this `cmp` style of comparisons is dropped, and **rich comparison** is introduced, which assign a special method to each operator as shown in Table. 15.2: To avoid the hassle of providing all six functions, we can only

Table 15.2: Operator and its special method

<code>==</code>	<code>__eq__</code>
<code>!=</code>	<code>__ne__</code>
<code><</code>	<code>__lt__</code>
<code><=</code>	<code>__le__</code>
<code>></code>	<code>__gt__</code>
<code>>=</code>	<code>__ge__</code>

implement `__eq__`, `__ne__`, and only one of the ordering operators, and use the `functools.total_ordering()` decorator to fill in the rest. For example, write a class **Person**:

```
1 from functools import total_ordering
2 @total_ordering
3 class Person(object):
4     def __init__(self, firstname, lastname):
5         self.first = firstname
```

```

6     self.last = lastname
7
8     def __eq__(self, other):
9         return ((self.last, self.first) == (other.last, other.
10        first))
11
12    def __ne__(self, other):
13        return not (self == other)
14
15    def __lt__(self, other):
16        return ((self.last, self.first) < (other.last, other.
17        first))
18
19    def __repr__(self):
20        return "%s %s" % (self.first, self.last)

```

Then, we would be able to use any of the above comparison operator on our class:

```

1 p1 = Person('Li', 'Yin')
2 p2 = Person('Bella', 'Smith')
3 p1 > p2

```

It outputs `True` because last name “Yin” is larger than “Smith”.

15.3 Naive Sorting

As the most naive and intuitive group of comparison-based sorting methods, this group takes $O(n^2)$ time and usually consists of two nested for loops. In this section, we learn three different sorting algorithms “quickly” due to their simplicity: insertion sort, bubble sort, and selection sort.

15.3.1 Insertion Sort

Insertion sort is one of the most intuitive sorting algorithms for humans. For humans, given an array of n items to process, we divide it into two regions: **sorted and unrestricted region**. Each time we take one item “out” of the unrestricted region to sorted region by inserting it at a proper position.

In-place Insertion The logic behind this algorithm is simple, we can do it easily by setting up another sorted array. However, here we want to focus on the in-place insertion. Given array of size n , we use index 0 and i to point to the start position of sorted and the unrestricted region, respectively. And $i = 1$ at the beginning, indicates that the sorted region will naturally has one item. We have sorted region in $[0, i - 1]$, and the unrestricted region in $[i, n - 1]$. We scan item in the unrestricted region from left to right, and insert each item $a[i]$ into the sorted sublist.

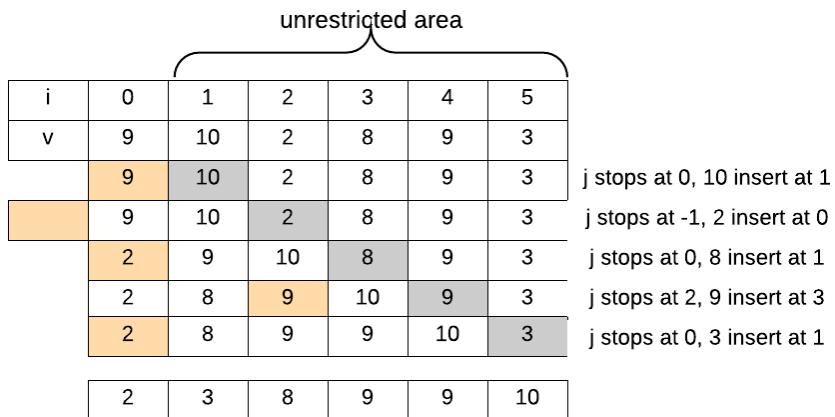


Figure 15.1: The whole process for insertion sort: Gray marks the item to be processed, and yellow marks the position after which the gray item is to be inserted into the sorted region.

The key step is to find a proper position of $a[i]$ in the region $[0, i - 1]$ to insert into. There are two different ways for iteration over unsorted region: forward and backward. We use pointer j in the sorted region.

- Forward: j will iterate in range $[0, i - 1]$. We compare $a[j]$ with $a[i]$, and stop at the first place that $a[j] > a[i]$ (to keep it stable). All items elements $a[j : i - 1]$ will be shifted backward for one position, and $a[i]$ will be placed at index j . Here we need i times of comparison and swaps.
- Backward: j iterates in range $[i - 1, 0]$. We compare $a[j]$ with $a[i]$, and stop at the first place that $a[j] \leq a[i]$ (to keep it stable). In this process, we can do the shifting simultaneously: if $a[j] > a[i]$, we shift $a[j]$ with $a[j + 1]$.

In forward, the shifting process still requires us to reverse the range, therefore the backward iteration makes better sense.

For example, given an array $a = [9, 10, 2, 8, 9, 3]$. First, 9 itself is sorted array. we demonstrate the backward iteration process. At first, 10 is compared with 9, and it stays at where it is. At the second pass, 2 is compared with 10, 9, and then it is put at the first position. The whole process of this example is demonstrated in Fig. 15.1.

With Extra Space Implementation The Python `list.insert()` function handles the insert and shifting at the same time. We need to pay attention when the item is larger than all items in the sorted list, we have to insert it at the end.

```

1 def insertionSort(a):
2     if not a or len(a) == 1:
3         return a
4     n = len(a)
5     sl = [a[0]] # sorted list
6     for i in range(1, n):
7         for j in range(i):
8             if sl[j] > a[i]:
9                 sl.insert(j, a[i])
10                break
11            if len(sl) != i + 1: # not inserted yet
12                sl.insert(i, a[i])
13

```

Backward In-place Implementation We use a `while` loop to handle the backward iteration: whenever the target is smaller than the item in the sorted region, we shift the item backward. When the `while` loop stops, it is either $j = -1$ or when $t \geq a[j]$.

- When $j = -1$, that means we need to insert the target at the first position which should be $j + 1$.
- When $t \geq a[j]$, we need to insert the target one position behind j , which is $j + 1$.

The code is shown as:

```

1 def insertionSort(a):
2     if not a or len(a) == 1:
3         return a
4     n = len(a)
5     for i in range(1, n):
6         t = a[i]
7         j = i - 1
8         while j >= 0 and t < a[j]:
9             a[j+1] = a[j] # Move item backward
10            j -= 1
11        a[j+1] = t
12

```

15.3.2 Bubble Sort and Selection Sort

Bubble Sort

Bubble sort compares each pair of adjacent items in an array and swaps them if they are out of order. Given an array of size n : in a single pass, there are $n - 1$ pairs for comparison, and at the end of the pass, one item will be put in place.

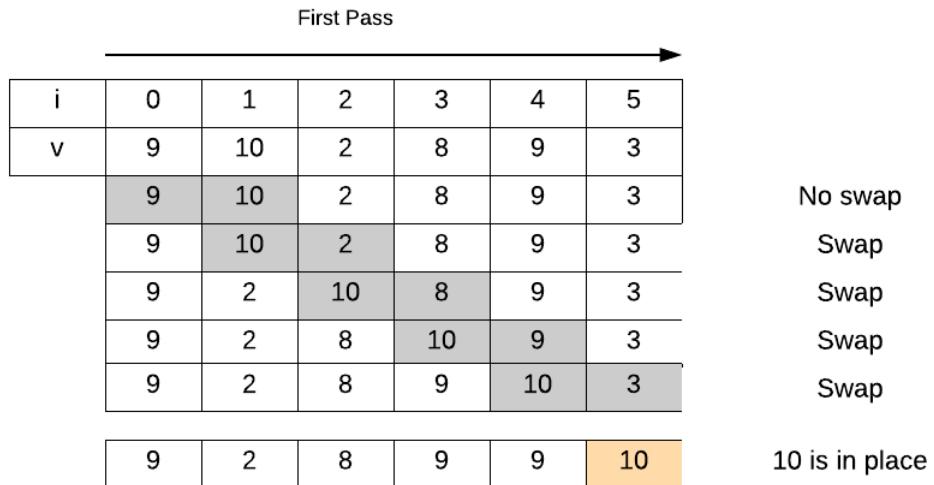


Figure 15.2: One pass for bubble sort

Passes For example, Fig. 15.2 shows the first pass for sorting array [9, 10, 2, 8, 9, 3]. When comparing a pair (a_i, a_{i+1}) , if $a_i > a_{i+1}$, we swap these two items. We can clearly see after one pass, the largest item 10 is in place. For the next pass, it only compare pairs within the unrestricted window $[0, 4]$. This is what “bubble” means in the name: after a pass, the largest item in the unrestricted window bubble up to the end of the window and become in place.

Implementation With the understanding of the valid window of each pass, we can implement “bubble” sort with two nested **for** loops in Python. The first **for** loop to enumerate the number of passes, say i , which is $n - 1$ in total. The second **for** loop to is to scan pairs in the unrestricted window $[0, n - i - 1]$ from left to right. thus index j points to the first item in the pair, making it in range of $[0, n - i - 2]$.

```

1 def bubbleSort(a):
2     if not a or len(a) == 1:
3         return
4     n = len(a)
5     for i in range(n - 1): #n-1 passes
6         for j in range(n - i - 1):
7             # Swap
8             if a[j] > a[j + 1]:
9                 a[j], a[j + 1] = a[j + 1], a[j]
10    return

```

When the pair has equal values, we do not need to swap them. The advantage of doing so is (1) to save unnecessary swaps and (2) keep the original order of items with same keys. This makes bubble sort a **stable sort**. Also,

in the implementation no extra space is assigned either which makes bubble sort **in-place sort**.

Complexity Analysis and Optimization In i -th pass, the item number in the valid window is $n - i$ with $n - i - 1$ maximum of comparison and swap, and we need a total of $n - 1$ passes. The total time will be $T = \sum_{i=0}^{n-i} (n - i - 1) = n - 1 + (n - 2) + \dots + 2 + 1 = n(n - 1)/2 = O(n^2)$. The above implementation runs $O(n^2)$ even if the array is sorted. We can optimize the inner **for** loop by stopping the whole program if no swap is detected in a single pass. When the input is nearly sorted, this strategy can get us $O(n)$ time complexity.

Selection Sort

i	0	1	2	3	4	5
v	9	10	2	8	9	3
	9	10	2	8	9	3
	9	3	2	8	9	10
	9	3	2	8	9	10
	8	3	2	9	9	10
	2	3	8	9	9	10
Sorted array						

Pass 1: swap 10 and 3
Pass 2: swap 9 with itself
Pass 3: swap 9 with 8
Pass 4: swap 8 with 2
Pass 5: swap 3 with 3

Figure 15.3: The whole process for Selection sort

In the bubble sort, each pass we get the largest element in the valid window in place by a series of swapping operations. While, selection sort makes a slight optimization via searching for the largest item in the current unrestricted window and swap it directly with the last item in the region. This avoids the constant swaps as occurred in the bubble sort. The whole sorting process for the same array is shown in Fig 15.3.

Implementation Similar to the implementation of Bubble Sort, we have the concept of number of passes at the outer **for** loop, and the concept of unrestricted at the inner **for** loop. We use variables ti and li for the position of the largest item to be and being, respectively.

```

1 def selectSort(a):
2     n = len(a)
3     for i in range(n - 1): #n-1 passes
4         ti = n - 1 - i
5         li = 0 # The index of the largest item

```

```

6     for j in range(n - i):
7         if a[j] >= a[li]:
8             li = j
9         # swap li and ti
10        a[ti], a[li] = a[li], a[ti]
11    return

```

Like bubble sort, selection sort is **in-place**. In the comparison, we used `if a[j] >= a[li]:`, which is able to keep the relative order of equal keys. For example, in our example, there is equal key 9. Therefore, selection sort is stable sort too.

Complexity Analysis Same as of bubble sort, selection sort has a worst and average time complexity of $O(n^2)$ but more efficient when the input is not as near as sorted.

15.4 Asymptotically Best Sorting

We have learned a few comparison-based sorting algorithms and they all have an upper bound of n^2 in time complexity due to the number of comparisons must be executed. Can we do better than $O(n^2)$ and how?

Comparison-based Lower Bounds for Sorting Given an input of size n , there are $n!$ different possible permutations on the input, indicating that our sorting algorithms must find the one and only one permutation by comparing pairs of items. So, how many times of comparison do we need to reach to the answer? Let's try the case when $n = 3$, and all possible permutations using the indexes will be: $(1, 2, 3), (1, 3, 2), (3, 1, 2), (2, 1, 3), (2, 3, 1), (3, 2, 1)$. First we compare pair $(1, 2)$, if $a_1 < a_2$, our candidates set is thus narrowed down to $\{(1, 2, 3), (1, 3, 2), (3, 1, 2)\}$.

We draw a decision-tree, which is a full binary tree with $n!$ leaves—the $n!$ permutations, and each branch represents one decision made on the comparison result. The cost of any comparison-based algorithm is abstracted as the length of the path from the root of the decision tree to its final sorted permutation. The longest path represents the worst-case number of comparisons.

Using h to denote the height of the binary tree, and l for the number of leaves. First, a binary tree will have at most 2^h leaves, we get $l \leq 2^h$. Second, it will have at least $n!$ leaves to represent all possible orderings, we have $l \geq n!$. Therefore we get the lower bound time complexity for the

worst case:

$$n! \leq l \leq 2^h \quad (15.1)$$

$$2^h \geq n! \quad (15.2)$$

$$h \geq \log(n!) \quad (15.3)$$

$$h = \Omega(n \log n) \quad (15.4)$$

In this section, we will introduce three classical sorting algorithms that has $O(n \log n)$ time complexity: Merge Sort and Quick Sort both utilize the Divide-and-conquer method, and Heap Sort uses the max/min heap data structures.

15.4.1 Merge Sort

As we know there are two main steps: “divide” and “merge” in merge sort and we have already seen the illustration of the “divide” process in Chapter. 13.

Divide In the divide stage, the original problem $a[s...e]$, where s, e is the start and end index of the subarray, respectively. The divide process divides its parent problem into two halves from the middle index $m = (s + e)//2$: $a[s...m]$, and $a[m + 1, e]$. This recursive call keeps moving downward till the size of the subproblem becomes one when $s = e$, which is the base case for a list of size 1 is naturally sorted. The process of divide is shown in Fig. 15.4.

Merge When we obtained two sorted sublists from the left and right side, the result of current subproblem is to merge the two sorted list into one. The merge process is done through two pointer method: We assign a new list and put two pointers at the start of the two sublists, and each time we choose the smaller item to append into the new list between the items indicated by the two pointers. Once a smaller item is chosen, we move its corresponding pointer to the next item in that sublist. We continue this process until any pointer reaches to the end. Then, the sublist where the pointer does not reach to the end yet is copied to the end of the new generated list. The subprocess is shown in Fig. 15.4 and its implementation is as follows:

```

1 def merge(l, r):
2     ans = []
3     # Two pointers each points at l and r
4     i = j = 0
5     n, m = len(l), len(r)
6
7     while i < n and j < m:
8         if l[i] <= r[j]:
9             ans.append(l[i])
10            i += 1

```

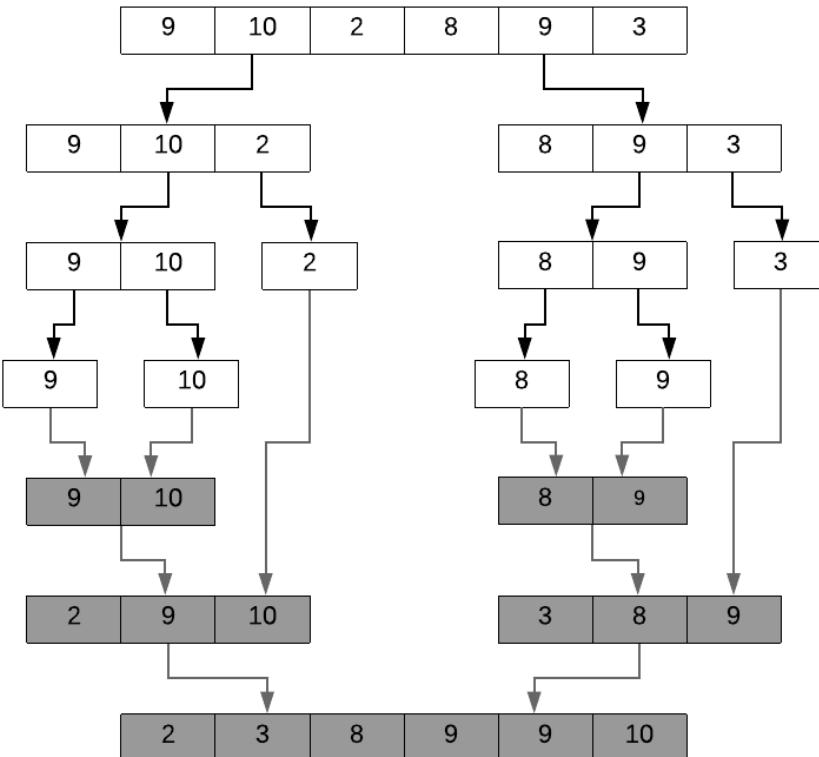


Figure 15.4: Merge Sort: The dividing process is marked with dark arrows and the merging process is with gray arrows with the merge list marked in gray color too.

```

11     else:
12         ans.append(r[j])
13         j += 1
14
15     ans += l[i:]
16     ans += r[j:]
17     return ans

```

In the code, we use $l[i] \leq r[j]$ instead of $l[i] < r[j]$ is because when the left and right sublist contains items of equal keys, we put the ones in the left first in the merged list, so that the sorting can be **stable**. However, we used a temporary space as $O(n)$ to save the merged result a , making merge sort an **out-of-place** sorting algorithm.

Implementation The whole implementation is straightforward.

```

1 def mergeSort(a, s, e):
2     if s == e:
3         return [a[s]]
4

```

```

5   m = (s + e) // 2
6
7   l = mergeSort(a, s, m)
8   r = mergeSort(a, m+1, e)
9   return merge(l, r)

```

Complexity Analysis Because for each divide process we need to take $O(n)$ time to merge the two sublists back to a list, the recurrent relation of the complexity function can be deducted as follows:

$$\begin{aligned}
 T(n) &= 2T(n/2) + O(n) \\
 &= 2 * 2T(n/4) + O(n) + O(n) \\
 &= O(n \log n)
 \end{aligned} \tag{15.5}$$

Thus, we get $O(n \log n)$ as the upper bound for merge sort, which is asymptotically optimal within the comparison-based sorting.

15.4.2 HeapSort

To sort the given array in increasing order, we can use min-heap. We first **heapify** the given array. To get a sorted list, we can simply pop out items till the heap is empty. And the popped out items will be in sorted order.

Implementation We can implement heap sort easily with built-in module `heapq` through the `heapify()` and `heappop()` functions :

```

1 from heapq import heapify, heappop
2 def heapsort(a):
3     heapify(a)
4     return [heappop(a) for i in range(len(a))]

```

Complexity Analysis The `heapify` takes $O(n)$, and the later process takes $O(\log n + \log(n-1) + \dots + 0) = \log(n!)$ which has an upper bound of $O(n \log n)$.

15.4.3 Quick Sort and Quick Select

Like merge sort, quick sort applies divide and conquer method and is mainly implemented with recursion. Unlike merge sort, the conquering step the sorting process- *partition* happens before “dividing” the problem into sub-problems through recursive calls.

Partition and Pivot In the partition, quick sort chooses a *pivot* item from the subarray, either randomly or intentionally. Given a subarray of $A[s, e]$, the pivot can either be located at s or e , or a random position in range $[s, e]$. Then it partitions the subarray $A[s, e]$ into three parts according to the value of the pivot: $A[s, p - 1]$, $A[p]$, and $A[p + 1 \dots e]$, where p is where the pivot is placed at. The left and right part of the pivot satisfies the following conditions:

- $A[i] \leq A[p], i \in [s, p - 1]$,
- and $A[i] > A[p], i \in [p + 1, e]$.

If we are allowed with linear space, this partition process will be trivial to implement. However, we should strive for better and learn an in-place partition methods—*Lomuto’s Partition*, which only uses constant space.

Conquer After the partition, one item—the pivot $A[p]$ is placed in the right place. Next, we only need to handle two subproblems: sorting $A[s, p - 1]$ and $A[p + 1, e]$ by recursively call the quicksort function. We can write down the main steps of quick sort as:

```

1 def quickSort(a, s, e):
2     # Base case
3     if s >= e:
4         return
5     p = partition(a, s, e)
6
7     # Conquer
8     quickSort(a, s, p-1)
9     quickSort(a, p+1, e)
10    return

```

At the next two subsection, we will talk about partition algorithm. And the requirement for this step is to do it **in-place** just through a series of swapping operations.

Lomuto’s Partition

We use example $A = [3, 5, 2, 1, 6, 4]$ to demonstrate this partition method. Assume our given range for partition is $[s, e]$, and $p = A[e]$ is chosen as pivot. We would use two pointer technique i, j to maintain three regions in subarray $A[s, e]$: (1) region $[s, i]$ with items smaller than or equal to p ; (2) $[i + 1, j - 1]$ region with item larger than p ; (3) unrestricted region $[j, e - 1]$. These three areas and the partition process on the example is shown in Fig. 15.5.

- At first, $i = s - 1, j = s$. Such initialization guarantees that region (1) and (2) are both empty, and region (3) is the full range other than the pivot.

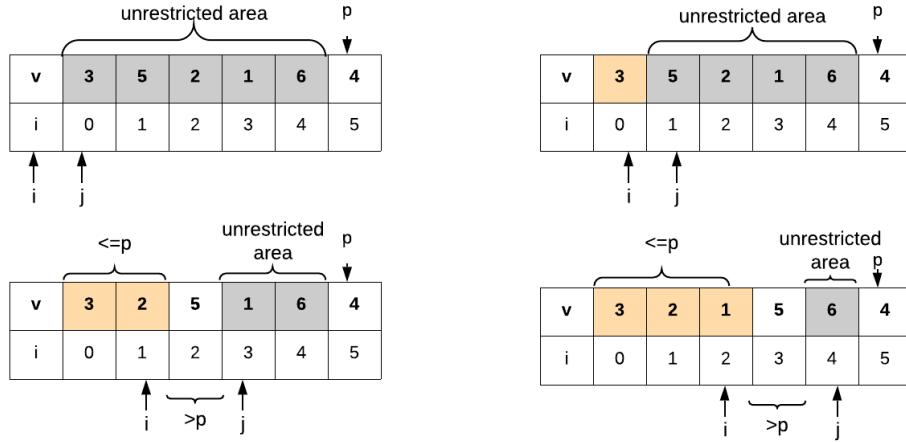


Figure 15.5: Lomuto’s Partition. Yellow, white, and gray marks as region (1), (2) and (3), respectively.

- Then, scan items in the unrestricted area using pointer j :
 - If the current item $A[j]$ belongs to region (2), that is to say $A[j] > p$, we just increment pointer j ;
 - Otherwise when $A[j] \leq p$, this item should go to region (1). We accomplish this by swapping this item with the first item in region (2) at $i + 1$. And now region (1) increments by one and region (2) shifts one position backward.
- After the for loop, we need to put our pivot at the first place of region (2) by swapping. And now, the whole subarray is successfully partitioned into three regions as we needed, and return where the index of where the pivot is at $-i + 1$ —as the partition index.

We The implementation of as follows:

```

1 def partition(a, s, e):
2     p = a[e]
3     i = s - 1
4     # Scan unrestricted area
5     for j in range(s, e):
6         # Swap
7         if a[j] <= p:
8             i += 1
9             a[i], a[j] = a[j], a[i]
10            a[i+1], a[e] = a[e], a[i+1]
11    return i+1

```

Complexity Analysis The worst case of the partition appears when the input array is already sorted or is reversed from the sorted array. In this

case, it will partition a problem with size n into one subproblem with size $n - 1$ and the other subproblem is just empty. The recurrence function is $T(n) = T(n-1) + O(n)$, and it has a time complexity of $O(n^2)$. And the best case appears when a subprocess is divided into half and half as in the merge sort, where the time complexity is $O(n \log n)$. Randomly picking the pivot from $A[s, e]$ and swap it with $A[e]$ can help us achieve a stable performance with average $O(n \log n)$ time complexity.

Stability of Quick Sort

Quick sort is *not stable*, because there are cases items can be swapped no matter what: (1) as the first item in the region (2), it can be swapped to the end of region (2). (2) as the pivot, it is swapped with the first item in the region (2) too. Therefore, it is hard to guarantee the stability among equal keys. We can try experiment with $A = [(2, 1), (2, 2), (1, 1)]$, and use the first element in the tuple as key. This will sort it as $A = [(1, 1), (2, 2), (2, 1)]$.

However, we can still make quick sort stable if we get rid of the swaps by using two extra lists: one for saving the smaller and equivalent items and the other for the larger items.

Quick Select

Quick Select is a variant of quick sort, and it is used to find the k -th smallest item in a list in linear time. In quicksort, it recurs both sides of the partition index, while in quick select, only the side that contains the k -th smallest item will be recurred. This is similar to the binary search, the comparison of k and partition index p results three cases:

- If $p = k$, we find the k -th smallest item, return.
- If $p > k$, then we recur on the right side.
- If $p < k$, then we recur on the left side.

Based on the structure, quick select has the following recurrence time complexity function:

$$T(n) = T(n/2) + O(n) \quad (15.6)$$

$$T(n) = O(n) \text{ (with master theorem)} \quad (15.7)$$

Implementation We first set k in range of $[s, e]$. When $s = e$, there is only one item in the list, which means we no longer can divide it. This is our end condition and is also the case when our original list has only one item, then we have to return this item as the 0-th smallest item.

```

1 def quickSelect(a, s, e, k, partition=partition):
2     if s >= e:
3         return a[s]
4
5     p = partition(a, s, e)
6     if p == k:
7         return a[p]
8     if k > p:
9         return quickSelect(a, p+1, e, k, partition)
10    else:
11        return quickSelect(a, s, p-1, k, partition)

```

15.5 Linear Sorting

Sorting without basing upon comparisons is possible, creative, and even faster, proved by the three non-comparative sorting algorithms we are about to introduce: Bucket Sort, Counting Sort, and Radix Sort. For these algorithms the theoretic lower bound $O(n \log n)$ of comparison-based sorting is not likewise a lower bound any more; they all work in linear time. However, there are limitations to the input data, as these sorting techniques rely on certain assumptions concerning the data to be sorted to be able to work.

Although the three algorithms we see in this section come in different forms and rely on different assumptions to the input data, we see one thing in common: They all use the divide and conquer algorithm design paradigm. Let's explore their unique tricks and the restrictive applications!

15.5.1 Bucket Sort

Bucket Sort assumes that the input data satisfying a uniform distribution. The uniform distribution is usually assumed to be in interval $[0, 1)$. However, it can be extended to any uniform distribution with simple modification. Bucket sort applies a one time divide and conquer trick—it divides the input data into n independent segments, n the size of the input, just as what we have seen in merge sort, and then insertion sort is applied on each segment, and finally each sorted segmented is combined to get the result.

Bucket sort manages the dividing process by assigning n empty **buckets**, and then distribute the input data $a[i]$ to bucket index `int(a[i]*n)`. For example, if $n = 10$, and $a[i] = 0.15$, the bucket that the number goes to is the one with index 1. We use example $a = [0.42, 0.72, 0., 0.3, 0.15, 0.09, 0.19, 0.35, 0.4, 0.54]$, and visualize the process in Fig. 15.6.

Implementation First, we prepare the input data with `random.uniform` from `numpy` library. For simplicity and the reconstruction of the same input, we used random seed and rounded the float number to only two decimals.

Input		Buckets			Sorted	
i	v	i		v	i	v
0	0.42	0	0.0	0.09	0	0
1	0.72	1	0.15	0.19	1	0.09
2	0	2			2	0.15
3	0.3	3	0.3	0.35	3	0.19
4	0.15	4	0.42	0.4	4	0.3
5	0.09	5	0.54		5	0.35
6	0.19	6			6	0.4
7	0.35	7	0.72		7	0.42
8	0.4	8			8	0.54
9	0.54	9			9	0.72

Figure 15.6: Bucket Sort

```

1 import numpy as np
2 np.random.seed(1)
3 a = np.random.uniform(0, 1, 10)
4 a = np.round(a, decimals=2)

```

Now, the code for the bucket sort is straightforward as:

```

1 from functools import reduce
2 def bucketSort(a):
3     n = len(a)
4     buckets = [[] for _ in range(n)]
5     # Divide numbers into buckets
6     for v in a:
7         buckets[int(v*n)].append(v)
8     # Apply insertion sort within each bucket
9     for i in range(n):
10        insertionSort(buckets[i])
11    # Combine sorted buckets
12    return reduce(lambda a, b: a + b, buckets)

```

Complexity Analysis

Extension To extend to uniform distribution in any range, we first find the minimum and maximum value, $\min V, \max V$, and compute the bucket

index i for number $a[i]$ with formula:

$$i = n \frac{a[i] - \min V}{\max V - \min V} \quad (15.8)$$

15.5.2 Counting Sort

Counting sort is an algorithm that sorts items according to their corresponding keys that are small integers. It works by counting the occurrences of each distinct key value, and using arithmetic-prefix sum-on those counts to determine the position of each key value in the sorted sequence. Counting sort no longer fits into the comparison-based sorting paradigm because it uses the keys as indexing to assist the sorting instead of comparing them directly to decide relative positions. For input that comes with size n and the difference between the maximum and minimum integer keys k , counting sort has a time complexity $O(n + k)$.

Premise: Prefix Sum

Before we introduce counting sort, first let us see what is prefix sum. Prefix sum, a.k.a cumulative sum, inclusive scan, or simply scan of a sequence of numbers $x_i, i \in [0, n - 1]$ is second sequence of numbers $y_i, i \in [0, n - 1]$, and y_i is the sums of prefixes of the input sequence, with equation:

$$y_i = \sum_{j=0}^i x_j \quad (15.9)$$

For instance, the prefix sums of on the following array is:

Index :	0	1	2	3	4	5
x :	1	2	3	4	5	6
y :	1	3	6	10	15	21

Prefix sums are trivial to compute with the following simple recurrence relation in $O(n)$ complexity.

$$y_i = y_{i-1} + x_i, i \geq 1 \quad (15.10)$$

Despite the ease of computation, prefix sum is a useful primitive in certain algorithms such as counting sort and Kadane's Algorithm as you shall see through this book.

Counting Sort

Given an input array $[1, 4, 1, 2, 7, 5, 2]$, let's see how exactly counting sort works by explaining it in three steps. Because our input array comes with duplicates, we distinguish the duplicates by their relative order shown in the parentheses. Ideally, for this input, we want it to be sorted as:

Index :	0	1	2	3	4	5	6
Key :	1(1)	4	1(2)	2(1)	7	5	2(2)
Sorted :	1(1)	1(2)	2(1)	2(2)	4	5	7

Input		Buckets		Prefix Sum	
i	v	key	count	key	count
0	1(1)	0	0	0	0
1	4	1	2	1	2
2	1(2)	2	2	2	4
3	2(1)	3	0	3	4
4	7	4	1	4	5
5	5	5	1	5	6
6	2(2)	6	0	6	6
		7	1	7	7

Figure 15.7: Counting Sort: The process of counting occurrence and compute the prefix sum.

1. **Count Occurrences:** We assign a `count` array C_i , and assign a size 8, which has index in range $[0, 7]$ and will be able to contain our keys whose range is $[1, 7]$. which has the same size of the key range k . We loop over each key in the input array, and use key as index to count each key's occurrence. Doing so will get the following result. And it means in the input array, we have two 1's, two 2's, one 4, one 5, and one 7. The process is shown in Fig. 15.7.

Counting sort is indeed a subtype of bucket sort, where the number of buckets is k , and each bucket stores keys implicitly by using keys as indexes and the occurrence to track the total number of the same keys.

2. **Prefix Sum on Count Array:** We compute the prefix sum for `count` array, which is shown as:

Index :	0	1	2	3	4	5	6	7
Count :	0	2	2	0	1	1	0	1
Prefix Sum:	0	2	4	4	5	6	6	7

Denote the prefix sum array as ps . For key i , ps_{i-1} tells us the number of items that is less or equals to (\leq) key i . This information can be

used to place key i directly into its correct position. For example, for key 2, summing over its previous keys' occurrences (ps_1) gives us 2, indicating that we can put key 2 to position 2. However, key 2 appears two times, and the last position of key 2 is indicated by $ps_2 - 1$, which is 3. Therefore, for any key i , its locations in the sorted array is in range $[ps_{i-1}, ps_i)$. We could have just scan the prefix sum array, and use the prefix sum as locations for key indicated by index of prefix sum array. However, this method is only limited to situations where the input array is integers. Moreover, it is unable to keep the relative ordering of the items of the same key.

3. Sort Keys with Prefix Sum Array: First, let us loop over the input keys from position 0 to $n - 1$. For key_i , we decrease the prefix sum by one, $ps_{key_i} = ps_{key_i} - 1$ to get the last position that we can assign this key in the sorted array. The whole process is shown in Fig. 15.8. We saw that items of same keys are sorted in reverse order. Looping over keys in the input in reverse order is able to correct this and thus making the counting sort a stable sorting algorithm.

	i	0	1	2	3	4	5	6									
	key	1(1)	4	1(2)	2(1)	7	5	2(2)	key	0	1	2	3	4	5	6	7
	ps	0	2	4	4	5	6	6	ps	0	2	4	4	5	6	6	7
1	key	1(1)	4	1(2)	2(1)	7	5	2(2)	key	0	1	2	3	4	5	6	7
1	sorted		1(1)						ps	0	2-1	4	4	5	6	6	7
2	key	1(1)	4	1(2)	2(1)	7	5	2(2)	key	0	1	2	3	4	5	6	7
2	sorted		1(1)				4		ps	0	1	4	4	5-1	6	6	7
3	key	1(1)	4	1(2)	2(1)	7	5	2(2)	key	0	1	2	3	4	5	6	7
3	sorted	1(2)	1(1)			4			ps	0	1-0	4	4	4	6	6	7
4	key	1(1)	4	1(2)	2(1)	7	5	2(2)	key	0	1	2	3	4	5	6	7
4	sorted	1(2)	1(1)		2(1)	4			ps	0	0	4-1	4	4	6	6	7
5	key	1(1)	4	1(2)	2(1)	7	5	2(2)	key	0	1	2	3	4	5	6	7
5	sorted	1(2)	1(1)		2(1)	4		7	ps	0	0	3	4	4	6	6	7-1
6	key	1(1)	4	1(2)	2(1)	7	5	2(2)	key	0	1	2	3	4	5	6	7
6	sorted	1(2)	1(1)		2(1)	4	5	7	ps	0	0	3	4	4	6-1	6	6
7	key	1(1)	4	1(2)	2(1)	7	5	2(2)	key	0	1	2	3	4	5	6	7
7	sorted	1(2)	1(1)	2(2)	2(1)	4	5	7	ps	0	0	3-1	4	4	5	6	6

Figure 15.8: Counting sort: Sort keys according to prefix sum.

Implementation In our implementation, we first find the range of the input data, say it is $[minK, maxK]$, making our range $k = maxK - minK$. And we recast the key as $key - minK$ for two purposes:

- To save space for `count` array.
- To be able to handle negative keys.

The implementation of the main three steps are nearly the same as what we have discussed other than the recast of the key. In the process, we used two auxiliary arrays: `count` array for counting and accumulating the occurrence of keys with $O(k)$ space and `order` array for storing the sorted array with $O(n)$ space, giving us the space complexity $O(n + k)$ in our implementation. The Python code is shown as:

```

1 def countSort(a):
2     minK, maxK = min(a), max(a)
3     k = maxK - minK + 1
4     count = [0] * (maxK - minK + 1)
5     n = len(a)
6     order = [0] * n
7     # Get occurrence
8     for key in a:
9         count[key - minK] += 1
10    # Get prefix sum
11    for i in range(1, k):
12        count[i] += count[i-1]
13
14    # Put key in position
15    for i in range(n-1, -1, -1):
16        key = a[i] - minK
17        count[key] -= 1 # to get the index as position
18        order[count[key]] = a[i]
19
20    return order

```

Properties Counting sort is **out-of-place** for the auxiliary `count` and `order` array. Counting sort is **stable** given that we iterate keys in the input array in reversed order. Counting sort is likely to have $O(n + k)$ for both the space and time complexity.

Applications Due to the special character that counting sort sorts by using key as index, and the range of keys decides the time and space complexity, counting sort's applications are limited. We list the most common applications:

- Because the time complexity depends on the size of k , in practice counting sort is usually used when $k = O(n)$, in which case it makes the time complexity $O(n)$.

- Counting sort is often used as a sub-routine. For example, it is a part of other sorting algorithms such as radix sort, which is a linear sorting algorithm. We will also see some examples in string matching chapter.

15.5.3 Radix Sort

The word “Radix” is a mathematical term for the *base* of a number. For example, decimal and hexadecimal number has a radix of 10 and 16, respectively. For strings of alphabets has a radix of 26 given there are 26 letters of alphabet. Radix sort is a non-comparative sorting methods that utilize the concept of radix or base to order a list of integers digit by digit or a list of strings letter by letter. The sorting of integers or strings of alphabets is different based on the different concepts of ordering—number ordering and the lexicographical order as we have introduced. We show one example for list of integers and strings and their sorted order or lexicographical order:

Integers :	170, 45, 75, 90, 802, 24
Sorted :	24, 45, 75, 90, 170, 802

Strings :	apple, pear, berry, peach, apricot
Sorted :	apple, apricot, berry, peach, pear

And we see how that the integers are ordered by the length of digits, whereas in the sorted strings, the length of strings does not usually decide the ordering.

Within Radix sorting, it is usually either the bucket sort or counting sort that is doing the sorting using one radix as key at a time. Based upon the sorting order of the digit, we have two types of radix sorting: *Most Significant Digit (MSD) radix sort* which starts from the left-most radix first and goes all the way the right-most radix, and *Least Significant Digit (LSD) radix sort* vice versa. We should address the details of the two forms of radix sort – MSD and LSD using our two examples.

LSD Radix Sorting Integers

LSD radix sort is often used to sort list of integers. It sorts the entire numbers one digit/radix at a time from the least-significant to the most-significant digit. For a list of positive integers where the maximum of them has m digits, LSD radix sort takes a total of m passes to finish sorting.

Here, we demonstrate the process of LSD radix sorting on our exemplary list of integers with counting sort as a subroutine to sort items by using each radix as key. $m = 3$ in our example:

- As shown in Fig. 15.9, in the first pass, the least significant digit (1st place) is used as key to sort. After this pass, the ordering of numbers of unit digits is in-place.

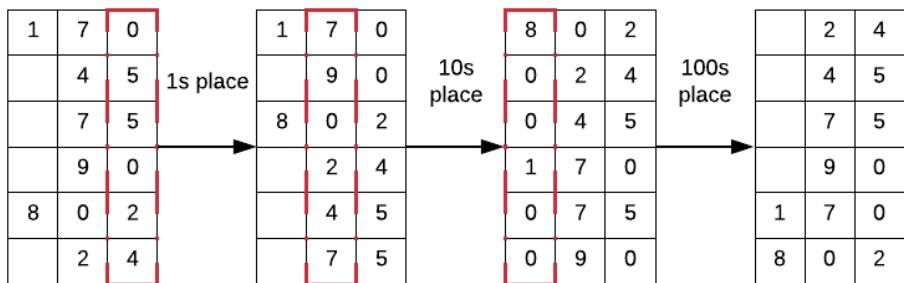


Figure 15.9: Radix Sort: LSD sorting integers in iteration

- In the second pass, the 10s place digit is used. After this pass, we see that numbers that has less than or equals to two digits comprising 24, 45, 75, 90 in our example is in ordering.
- At the last and third pass, the 100s place digit is used. For numbers that are short of 100s place digit, 0 is placed. Afterwards, the entire numbers are in ordering.

We have to notice that the sorting will not work unless the sorting subroutine we apply is stable. For example, in our last pass, there exists four zeros, indicating that they share the same key value. If the relative ordering of them is not kept, the previously sorting effort will be wasted.

Implementation To implement the code with Python, we first need to know how to get each digit out of an integer. With number 178 as an example:

- The least significant digit 8 is the remainder of $178 \% 10$.
- The second least-significant digit 7 is the remainder of $17 \% 10$.
- And the most-significant digit 1 is the remainder of $1 \% 10$.

As we see for digit 8, we need to have 178, for digit 7, we need to have 17, and for digit 1, we only need 1. 178, 17, 1 are the prefix till the digit we need. We can obtain these prefixes via a base exp .

```
exp = 1, (178 // exp) = 178, 178 % 10 = 8
exp = 10, (178 // exp) = 17, 17 % 10 = 7
exp = 100, (178 // exp) = 1, 1 % 10 = 1
```

We can also get the prefix by looping and each time we divide our number by 10. For example, the following code will output [8, 7, 1].

```
1 a = 178
2 digits = []
```

```

3 while a > 0:
4     digits.append(a%10)
5     a = a // 10

```

Now, we know the number of loops we need is decided by the maximum positive integer in our input array. On the code basis, we use a `while` loop to obtain the prefix and making sure that it is larger than 0. At each pass, we call `count_sort` subroutine to sort the input list. The code is shown as:

```

1 def radixSort(a):
2     maxInt = max(a)
3     exp = 1
4     while maxInt // exp > 0:
5         a = count_sort(a, exp)
6         exp *= 10
7     return a

```

For subroutine `count_sort` subroutine, it is highly similar to our previously implemented counting sort but two minor differences:

- Because we sort by digits, therefore, we have to use a formula: $key = (key//exp)\%10$ to covert the key to digit.
- Because for decimal there are in total only 10 digits, we only arrange 10 total space for the `count` array.

The code is as:

```

1 def count_sort(a, exp):
2     count = [0] * 10 # [0, 9]
3     n = len(a)
4     order = [0] * n
5     # Get occurrence
6     for key in a:
7         key = (key // exp) % 10
8         count[key] += 1
9
10    # Get prefix sum
11    for i in range(1, 10):
12        count[i] += count[i-1]
13
14    # Put key in position
15    for i in range(n-1, -1, -1):
16        key = (a[i] // exp) % 10
17        count[key] -= 1 # to get the index as position
18        order[count[key]] = a[i]
19    return order

```

Properties and Complexity Analysis Radix sorting for integers takes m passes with m as the total digits, and each pass takes $O(n + k)$, where $k = 10$ since there is only 10 digits for decimals. This gives out a total of $O(mn)$ time complexity, and m is rather of a constant compared with

variable n , thus radix sorting for integers with counting sort as subroutine has a linear time complexity. Due to the usage of counting sort, which is stable, making the radix sorting a stable sorting algorithm too.

With the usage of auxiliary `count` and `order`, it gives a $O(n)$ space complexity, and makes the LSD integer sorting an out-of-place sorting algorithm.

MSD Radix Sorting Strings

In our fruit alphabetization example, it uses MSD radix sorting and groups the strings by a single letter with either bucket sort or counting sort under the hood, starting from the very first letter on the left side all the way to the very last on the right if necessary. MSD radix sorting is usually implemented with recursion.

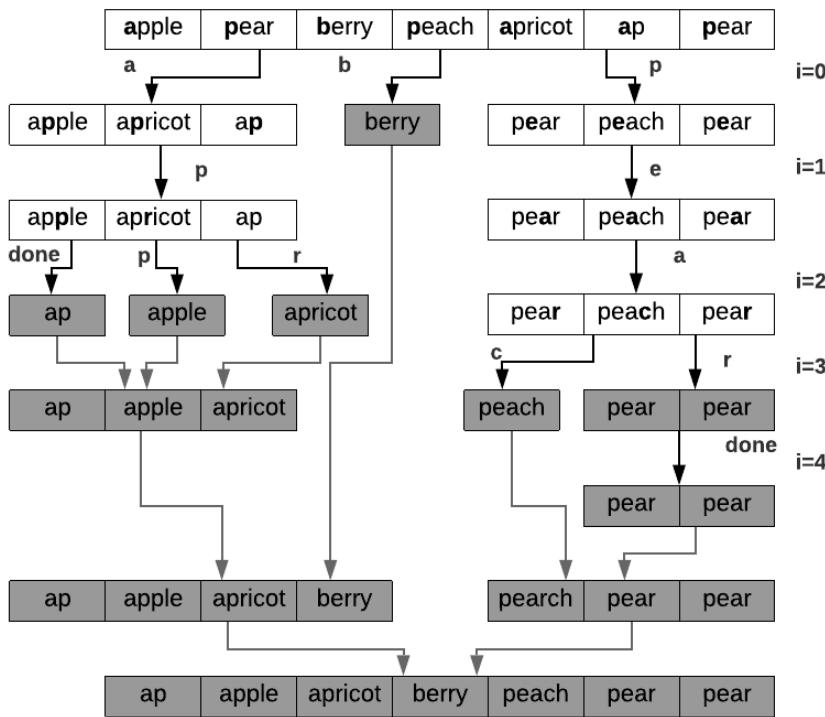


Figure 15.10: Radix Sort: MSD sorting strings in recursion. The black and grey arrows indicate the forward and backward pass in recursion, respectively.

For better demonstration, we add two more strings: “ap” and “pear”. String “ap” is for showing what happens when the strings in the same bucket but one is shorter and has no valid letter to compare with. And string “pear” is to showcase how the algorithm handles duplicates. The algorithm indeed

applies the recursive divide and conquer design methodology.

Implementation We show bucket sort here, as in lexicographical ordering, the first letter of the strings already decide the ordering of groups of strings bucketed by this letter. This LSD sorting method divide the strings into different buckets indexed by letter, and it then combines the returned results together to get the final sorted strings, which is highly similar to merge sort.

- At first, the recursion handles the first keys of the first letter in the string, as the process where $i = 0$ shown in Fig. 15.10. There are three buckets with letter ‘a’, ‘b’, and ‘p’.
- At depth 2 when $i = 1$, the resulting buckets from depth 1 is further bucketed by the second letter. Bucket ‘b’ contains only one item, which itself is sorted, thus, the recursion end for this bucket. For the last bucket ‘a’ and ‘p’, they are further bucketed by letter ‘p’ and ‘e’.
- At depth 3 when $i = 2$, for the last bucket ‘p’, it further results two more buckets ‘p’ and ‘r’. However, for string ‘ap’ in the bucket, there is no valid third letter to use as index. And according to the lexicographical order, it puts ‘ap’ in earlier ordering of the resulted buckets.
- In our example, the forward process in the recursion is totally done when $i = 4$. It then enters into the backward phase, which merges buckets that are either composed of a single item or the `done_bucket`.

The code is offered as:

```

1 def MSD_radix_string_sort(a, i):
2     # End condition: bucket has only one item
3     if len(a) <= 1:
4         return a
5
6     # Divide
7     buckets = [[] for _ in range(26)]
8     done_bucket = []
9     for s in a:
10        if i >= len(s):
11            done_bucket.append(s)
12        else:
13            buckets[ord(s[i]) - ord('a')].append(s)
14    # Conquer and chain all buckets
15    ans = []
16    for b in buckets:
17        ans += MSD_radix_string_sort(b, i + 1)
18    return done_bucket + ans

```

Properties and Complexity Analysis Because the bucket sort itself is a stable sorting algorithm, making the radix sort for string stable too.

The complexity analysis for the recursive Radix sorting can be accomplished with recursion tree. The tree has nearly n leaves. The worst case occurs when all strings within the input array are the same, thus the recursion tree degrades to linear structure with length n and within each node $O(n)$ is spent to scan items of corresponding letter, making the worst time complexity $O(n^2)$.

For the existence of auxiliary `buckets`, `done_bucket`, and `ans` arrays in sorting of strings, it is an out-of-place sorting. With the same recursion tree analysis for space, we have linear space complexity too.

15.6 Python Built-in Sort

There are two built-in functions to sort `list` and other iterable objects in Python 3, and both of them are stable. In default, they use `<` comparisons between items and sort items in increasing order.

- Built-in method `list.sort(key=None, reverse=False)` of `list` which sorts the items in the list in-place, and returns `None`.
- Built-in function `sorted(iterable, key=None, reverse=False)` works on *any iterable object*, including `list`, `string`, `tuple`, `dict`, and so on. It sorts the items out-of-place; returning another `list` and keeps the original input unmodified.

Basics

To use the above two built-in methods to sort a list of integers is just as simple as:

```
1 lst = [4, 5, 8, 1, 2, 7]
2 lst.sort()
```

Printing out `lst` shows that the sorting happens in-place within `lst`.

```
1 [1, 2, 4, 5, 7, 8]
```

Now, use `sorted()` for the same list:

```
1 lst = [4, 5, 8, 1, 2, 7]
2 new_lst = sorted(lst)
```

We print out:

```
1 new_lst, lst
2 ([1, 2, 4, 5, 7, 8], [4, 5, 8, 1, 2, 7])
```

Let's try to sort other iterable object, and try sort a tuple of strings:

```
1 fruit = ('apple', 'pear', 'berry', 'peach', 'apricot')
2 new_fruit = sorted(fruit)
```

Print out `new_fruit`, and we also see that it returned a `list` instead of `tuple`.

```
1 ['apple', 'apricot', 'berry', 'peach', 'pear']
```

Note: For `list`, `list.sort()` is faster than `sorted()` because it doesn't have to create a copy. For any other iterable, we have no choice but to apply `sorted()` instead.

Change Comparison Operator What if we want to redefine the behavior of comparison operator `<`? Other than writing a class and defining `__lt__()`, in Python 2, these two built-in functions has another argument, `cmp`, but it is totally dropped in Python 3. We can use `functools`'s `cmp_to_key` method to convert to `key` in Python 3. For example, we want to sort `[4, 5, 8, 1, 2, 7]` in reverse order, we can define a `cmp` function that reverse the order of items to be compared:

```
1 def cmp(x, y):
2     return y - x
```

And then we call this function as:

```
1 from functools import cmp_to_key
2 lst.sort(key=cmp_to_key(cmp))
```

The printout of `lst` is:

```
1 [8, 7, 5, 4, 2, 1]
```

Timsort These two methods both using the same sorting method – *Tim-sort* and has the same parameters. Timesort is a hybrid stable and in-place sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", January 1993. It was implemented by Tim Peters in 2002 for use in the Python programming language. The algorithm finds subsequences of the data that are already ordered, and uses that knowledge to sort the remainder more efficiently.

Arguments

They both takes two keyword-only arguments: `key` and `reverse`, and each has `None` and `False` as default value, respectively.

- Argument `key`: ot specifies a function of one argument that is used to extract a comparison key from each list item (for example, `key=str.lower`). If not set, the default value `None` means that the list items are sorted directly.

- Argument `reverse`: a boolean value. If set to `True`, then the list or iterable is sorted as if each comparison were reversed(use `>=` sorted list is in Descending order. The default value is `False`.

Sort in Reverse Order Set `reverse=True` will sort a list of integers in decreasing order:

```
1 lst = [4, 5, 8, 1, 2, 7]
2 lst.sort(reverse=True)
```

Print out `lst`, we see:

```
1 [8, 7, 5, 4, 2, 1]
```

This is equivalent to customize a class `Int` and rewrite its `__lt__()` special method as:

```
1 class Int(int):
2     def __init__(self, val):
3         self.val = val
4     def __lt__(self, other):
5         return other.val < self.val
```

Now, sort the same list but without setting `reverse` will get us exactly the same result:

```
1 lst = [Int(4), Int(5), Int(8), Int(1), Int(2), Int(7)]
2 lst.sort()
```

Customize key We have mainly two options to customize the `key` argument: (1) through `lambda` function, (2) through a pre-defined function. And in either way, the function only takes one argument. For example, to sort the following list of tuples by using the second item in the tuple as key:

```
1 lst = [(8, 1), (5, 7), (4, 1), (1, 3), (2, 4)]
```

We can write a function, and set `key` argument to this function

```
1 def get_key(x):
2     return x[1]
3 new_lst = sorted(lst, key = get_key)
```

The sorted result is:

```
1 [(8, 1), (4, 1), (1, 3), (2, 4), (5, 7)]
```

The same result can be achieved via `lambda` function which is more convenient:

```
1 new_lst = sorted(lst, key = lambda x: x[1])
```

Same rule applies to objects with named attributes. For example, we have the following class named `Student` that comes with three attributes: `name`, `grade`, and `age`, and we want to sort a list of `Student` class only through the `age`.

```

1 class Student(object):
2     def __init__(self, name, grade, age):
3         self.name = name
4         self.grade = grade
5         self.age = age
6
7     # To support indexing
8     def __getitem__(self, key):
9         return (self.name, self.grade, self.age)[key]
10
11    def __repr__(self):
12        return repr((self.name, self.grade, self.age))

```

We can do it through setting `key` argument still:

```

1 students = [Student('john', 'A', 15), Student('jane', 'B', 12),
2           Student('dave', 'B', 10)]
2 sorted(students, key=lambda x: x.age)

```

which outputs the following result:

```
1 [( 'dave', 'B', 10), ( 'jane', 'B', 12), ( 'john', 'A', 15)]
```

The key-function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The `operator` module has `itemgetter()` and `attrgetter()` to get the attributes by index and name, respectively. For the sorting above, we can do it like this:

```

1 from operator import attrgetter
2 sorted(students, key=attrgetter('age'))

```

`attrgetter` can take multiple arguments, for example, we can sort the list first by ‘grade’ and then by ‘age’, we can do it as:

```
1 sorted(students, key=attrgetter('grade', 'age'))
```

which outputs the following result:

```
1 [( 'john', 'A', 15), ( 'dave', 'B', 10), ( 'jane', 'B', 12)]
```

If our object supports indexing, which is why we defined `__getitem__()` in the class, we can use `itemgetter()` to do the same thing:

```

1 from operator import itemgetter
2 sorted(students, key=itemgetter(2))

```

15.7 Summary and Bonus

Here, we give a comprehensive summary of the time complexity for different sorting algorithms.

	Worst Case	Average Case	Best Case
Bubble Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Insertion Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Heap Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

Figure 15.11: The time complexity for common sorting algorithms

15.8 LeetCode Problems

Problems

15.1 Insertion Sort List (147). Sort a linked list using insertion sort.

A graphical example of insertion sort. The partial sorted list (black) initially contains only the first element in the list. With each iteration one element (red) is removed from the input data and inserted in-place into the sorted list

Algorithm of Insertion Sort: Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Example 1:

Input: 4->2->1->3

Output: 1->2->3->4

Example 2:

Input: -1->5->3->4->0

Output: -1->0->3->4->5

15.2 Merge Intervals (56, medium). Given a collection of intervals, merge all overlapping intervals.

Example 1:

Input: [[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].

Example 2:

Input: $[[1, 4], [4, 5]]$

Output: $[[1, 5]]$

Explanation: Intervals $[1, 4]$ and $[4, 5]$ are considered overlapping.

- 15.3 **Valid Anagram (242, easy).** Given two strings s and t , write a function to determine if t is an anagram of s .

Example 1:

Input: $s = \text{"anagram"}, t = \text{"nagaram"}$

Output: true

Example 2:

Input: $s = \text{"rat"}, t = \text{"car"}$

Output: false

Note: You may assume the string contains only lowercase alphabets.

Follow up: What if the inputs contain unicode characters? How would you adapt your solution to such case?

- 15.4 **Largest Number (179, medium).**

- 15.5 **Sort Colors (leetcode: 75).** Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue. Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively. *Note: You are not suppose to use the library's sort function for this problem.*

- 15.6 **148. Sort List (sort linked list using merge sort or quick sort).**

Solutions

1. Solution: the insertion sort is easy, we need to compare current node with all previous sorted elements. However, to do it in the linked list, we need to know how to iterate elements, how to build a new list. In this algorithm, we need two while loops to iterate: the first loop go through from the second node to the last node, the second loop go through the whole sorted list to compare the value of the current node to the sorted element, which starts from having one element. There are three cases for the comparison: if the comp_node does not move, which means we need to put the current node in front the previous head, and the cur_node become the new head; if the comp_node stops at the back of it, so current node is the end, we set its value to 0, and we save the pre_node in case; if it stops in the middle, we need to put cur_node in between pre_node and cur_node.

```

1 def insertionSortList(self, head):
2     """
3     :type head: ListNode
4     :rtype: ListNode
5     """
6     if head is None:
7         return head
8     sorted_head = head
9     cur_node = head.next
10    head.next = None #sorted list only has one node, a new
11    list
12    while cur_node:
13        next_node = cur_node.next #save the next node
14        cmp_node = head
15        #compare node with previous all
16        pre_node = None
17        while cmp_node and cmp_node.val <= cur_node.val:
18            pre_node = cmp_node
19            cmp_node = cmp_node.next
20
21        if cmp_node == head: #put in the front
22            cur_node.next = head
23            head = cur_node
24        elif cmp_node == None: #put at the back
25            cur_node.next = None #current node is the end,
26            so set it to None
27            pre_node.next = cur_node
28            #head is not changed
29        else: #in the middle, insert
30            pre_node.next = cur_node
31            cur_node.next = cmp_node
32            cur_node = next_node
33
34    return head

```

2. Solution: Merging intervals is a classical case that use sorting. If we do the sorting at first, and keep track our merged intervals in a heap (which itself its sorted too), we just iterate into the sorted intervals, to see if it should be merged in the previous interval or just be added into the heap. Here the code is tested into Python on the Leetcode, however for the python3 it needs to resolve the problem of the heappush with customized class as iterable item.

```

1 # Definition for an interval.
2 # class Interval(object):
3 #     def __init__(self, s=0, e=0):
4 #         self.start = s
5 #         self.end = e
6 from heapq import heappush, heappop
7
8 class Solution(object):
9     def merge(self, intervals):
10         """
11             :type intervals: List[Interval]

```

```

12     :rtype: List[Interval]
13     """
14     if not intervals:
15         return []
16     #sorting the intervals nlogn
17     intervals.sort(key=lambda x:(x.start , x.end))
18     h = [intervals[0]]
19     # iterate the intervals to add
20     for i in intervals[1:]:
21         s , e = i.start , i.end
22         bAdd = False
23         for idx , pre_interal in enumerate(h):
24             s_before , e_before = pre_interal.start ,
25             pre_interal.end
26             if s <= e_before: #overlap , merge to the
27                 same interval
28                 h[idx].end = max(e , e_before)
29                 bAdd = True
30                 break
31             if not bAdd:
32                 #no overlap , push to the heap
33                 heappush(h, i)
34     return h

```

3. Solution: there could have so many ways to do it, the most easy one is to sort the letters in each string and see if it is the same. Or we can have an array of 26, and save the count of each letter, and check each letter in the other one string.

```

1 def isAnagram(self , s , t):
2     """
3     :type s: str
4     :type t: str
5     :rtype: bool
6     """
7     return ''.join(sorted(list(s))) == ''.join(sorted(
8         list(t)))

```

The second solution is to use a fixed number counter.

```

1 def isAnagram(self , s , t):
2     """
3     :type s: str
4     :type t: str
5     :rtype: bool
6     """
7     if len(s) != len(t):
8         return False
9     table = [0]*26
10    start = ord('a')
11    for c1 , c2 in zip(s , t):
12        print(c1 , c2)
13        table[ord(c1)-start] += 1
14        table[ord(c2)-start] -= 1

```

```

15     for n in table:
16         if n != 0:
17             return False
18     return True

```

For the follow up, use a hash table instead of a fixed size counter. Imagine allocating a large size array to fit the entire range of unicode characters, which could go up to more than 1 million. A hash table is a more generic solution and could adapt to any range of characters.

4. Solution: from instinct, we know we need sorting to solve this problem. From the above example, we can see that sorting them by integer is not working, because if we do this, with 30, 3, we get 303, while the right answer is 333. To review the sort built-in function, we need to give a key function and rewrite the function, to see if it is larger, we compare the concatenated value of a and b, if it is larger. The time complexity here is $O(n \log n)$.

```

1 class LargerNumKey(str):
2     def __lt__(x, y):
3         return x+y > y+x
4
5 class Solution:
6     def largestNumber(self, nums):
7         largest_num = ''.join(sorted(map(str, nums), key=
LargerNumKey))
8         return '0' if largest_num[0] == '0' else
largest_num

```


16

Dynamic Programming

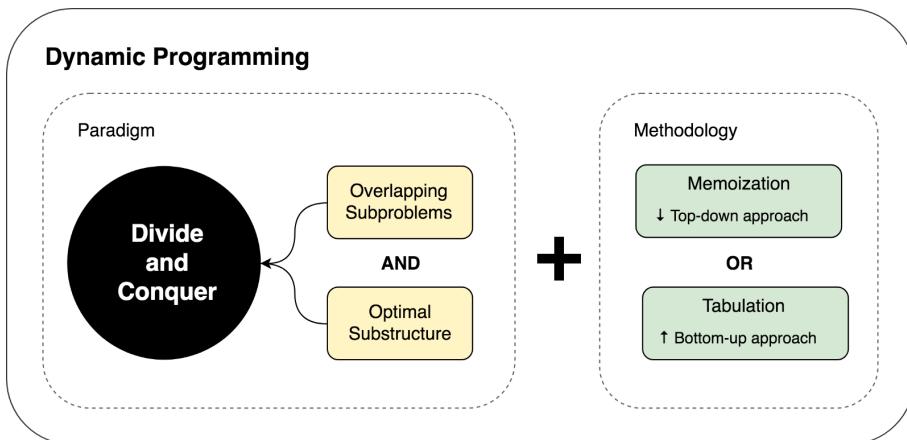


Figure 16.1: Dynamic Programming Chapter Recap

Dynamic programming is simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. As introduced in Divide-and-Conquer in Chapter 13, dynamic programming is applied on problems wherein its subproblems overlap when you construct them in Divide-and-Conquer manner. We use the recurrence function: $T(n) = T(n - 1) + T(n - 2) + \dots + T(1) + f(n)$, to highlight its most special characteristic – Overlapping Subproblems – compared with $T(n) = 2T(n/2) + f(n)$ for Divide-and-Conquer's nonoverlapping subproblems. As we shall see there are more types of recurrence functions in dynamic programming field other than this exemplary formulation, either in this chapter briefly or in Chapter 15 where a comprehensive list of dynamic programming categories/patterns.

terns are given.

Importance and Applications Dynamic Programming is one of the fundamental methods in computer science and plays a very important role in computer algorithms, even the book *artificial Intelligence, a modern approach* has mentioned this terminology for as many as 47 times. Dynamic programming first for optimizing the problems fall into the following categories:

1. Optimization: Compute the maximum or minimum value;
2. Counting: Count the total number of solutions;
3. Checking if a solution works.

To be noticed, not that all problems with the above formats will be certainly solved with dynamic programming, it requires the problem to show two properties: overlapping subproblems and optimal substructures in order for dynamic programming to be applied. These two properties will be defined and explained in this chapter.

Our Difference and Plan A lot of textbooks or courses describe dynamic programming as obscure and demands creativity and subtle insights from users to identify and construct its dynamic programming solutions. However, we are determined to unfold such “mystery” by being grounding practical. We have two chapters topiced with dynamic programming: The current chapter oriented with clear definition by distinguishing, relating, and exemplifying the concept with divide-conquer and complete search. This chapter serves as the frontline of our contents on Dynamic Programming. Further, Chapter 15 is focusing on categorizing problems patterns and giving examples on each.

- In order to understand how dynamic programming’s role in the algorithms evolution map, the very first thing we do in Section 16.1 is to show how we evolve the complete search to dynamic programming solution by: (1) discussing the relation between complete search, divide-and-conquer, and our dynamic programming; and (2) examining two elementary examples – Fibonacci Sequence and Longest Increasing Subsequence.
- Dynamic programming is typically applied on optimization problems. Section 16.2 discuss the principle properties, elements, and experience based guideline. And we show how we can relate these key characteristics to the field of optimization.

- The naive solutions for dynamic programming applicable problems have either exponential or polynomial time using complete searching method. In Section 16.3 we showcase how we can decrease the complexity from the two baselines: from exponential to polynomial and from polynomial to polynomial with lower power.

16.1 Introduction to Dynamic Programming

In this section, we answer two questions:

- How to distinct divide and conquer from dynamic programming?** We have already conceptually know that it differs in the case of the characteristics of subproblems in two cases: overlapping subproblems for dynamic programming where each subproblems share subproblems and non-overlapping subproblems for divide and conquer where the subproblems are disjoint with each other. In this section, we further answer this question in a more visualized way using the concept of *subproblem graph*.
- How to develop the dynamic programming solution from the complete search naive method?** We are not offering a fully and detail-oriented answer in this section. Instead, we first identify the problem using complete search in dynamic programming applicable problems using subproblem graph. Then we answer this question using two elementary examples by showing a sorted solutions so that we can demonstrate the relation between complete search and dynamic programming.

16.1.1 Concepts

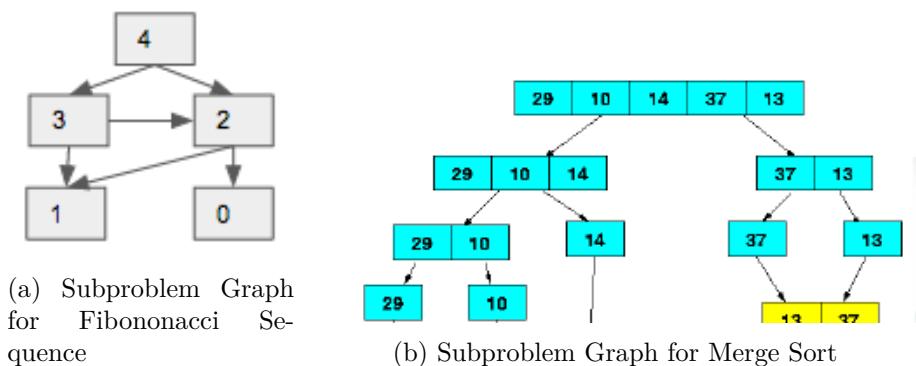


Figure 16.2: Subproblem Graph

Subproblem Graph If we treat each subproblem as a vertex, and the relation between subproblems as arced edges, we can get a directed subproblem graph. If the arced edge points from larger subproblems to smaller subproblems, we say it is in *top-down fashion*. In contrast, if arced edge is pointing from smaller subproblems to large subproblems, it is *bottom-up fashion*. In Fig. 16.2 we draw the subproblem graph for Fibonacci Sequence which we have defined in Page?? with $n = 4$. In comparison, we also give the subproblem graph for sorting for array [29, 10, 14, 37, 13]. In the following contents, we show how we can use subproblem graph to answer the two questions we lay out at the beginning of this section.

Terminologies To make reading other materials accessible, we introduce more related terminologies that are widely used in the field.

- **State:** State and subproblem is interchangeable among different books. Both of them can be used to describe the optimal solution to a problem with a solution space/search space.
- **State Transition:** State transition and recurrence function is interchangeable. In the case of fibonacci sequence, our state transition/recurrence function is given as $f(n) = f(n - 1) + f(n - 2)$. On the flip side, there are problems that will not specify the state transitions. We will have to figure them out by ourselves.

Distinction between Divide and Conquer and Dynamic Programming In divide and conquer, problems are divided into disjoint subproblems, the subproblem graph would degrade to a *tree structure*, each subproblem other than the base problems (here it is each individual element in the array) will only have out degree equals to 1. In comparison, in the case of fibonacci sequence shown in Fig. 16.2), some problems would have out degree larger than one, which makes the network a graph instead of a tree structure. This characteristics is directly induced by the fact that the subproblems overlap – that is, subproblems share subproblems. For example, $f(4)$ and $f(3)$ share $f(2)$, and $f(3)$ and $f(2)$ shares $f(1)$.

Complete Search and Dynamic Programming If we program these two problems in a recursive top-down divide-and-conquer manner, they are essentially equivalent to applying Depth-first-search on the subproblem graph/tree. The only difference is for sorting, there will be no recomputation (such as in merge sort and quick sort), while for the fibonacci sequence, subproblems that have in-degree larger than 1 will be recomputed multiple times, which gives us space for optimization and this is where the dynamic programming comes into rescue.

With the subproblem graph, we reconstruct the problem as a graph problem, which means all complete searching methods can be applied; such as Breadth-first search other than the recursive depth-first-search.

Depth-first-search Because of the usage of subproblems, the depth-first-search implemented with divide-and-conquer manner (with the result of subproblems as return) outweigh the usage of Breath-first-search. BFS doesn't compute the values of "optimal" sub-solutions (of sub-instances) and use these to build up a solution, then build the optimum using this information. I don't see anything "bottom-up" in the process of BFS. I don't see where the "intermediate" states come. Therefore, we shall see that the close bond of dynamic programming with DFS instead of with BFS.

16.1.2 From Complete Search to Dynamic Programming

So far, we know dynamic programming is an optimization methodology over the compete search solutions for typical optimization problems. Dynamic Programming's core principle is to solve each subproblem only *once* by *saving* and *reusing* its solution. Therefore, compare with its naive counterpart – Complete Search:

1. Dynamic Programming avoids the redundant recomputation met in its compete search counterpart as demonstrated in the last section.
2. Dynamic Programming uses additional memory as a trade-off for better computation/time efficiency; it serves as an example of a *time-space trade-off*. In most cases as we shall see in this chapter, the space overhead is well-worth; it can decrease the time complexity dramatically from exponential to polynomial level.

Two Forms of Dynamic Programming Solution There are two ways – either recursive or iterative in general to add *space mechanism* into naive complete search to construct our dynamic programming solution. But do remember that we cannot eliminate recursive thinking completely. we will always have to define a recursive relation irrespective of the approach we use.

1. **Top-down + Memoization (recursive DFS):** we start from larger problem (from top) and recursively search the subproblems space (to bottom) until the leaf node. This method is built on top of Depth-First Graph Search together with Divide and Conquer Methodology which treat each node as a subproblem and return its solution to its caller so that it can be used to build up its solution. Following a top-down fashion as is in divide and conquer, along the process, in the recursive call procedure, a hashmap is relied on to save and search solutions.

The memoization works in such way that at the very first time that the subproblem is solved it will be saved in the hashmap, and whenever this problem is met again, it finds the solution and returns it directly instead of computing again. The key elements of this style of dynamic programming is:

- (a) Define subproblem;
 - (b) Develop solution using Depth-first Graph search and Divide and conquer (leave alone the recomputation).
 - (c) Adding hashmap to save and search the state of each subproblem.
2. **Bottom-up + Tabulation (iterative):** different from the last method, which use recursive calls, in this method, we approach the subproblems from the smallest subproblems, and construct the solutions to larger subproblems using the tabulated result. The nodes in the subproblem graph is visited in a *reversed topological sort order*. This means that to reconstruct the state of current subproblem, all dependable (predecessors) have already be computed and saved.

Comparison The Figure 16.1 record the two different methods, we can use *memoization* and *tabulation* for short. Memoization and tabulation yield the same asymptotic time complexity, however the tabulation approach often has much better constant factors, since it has less overhead for procedure calls.

The memoization method applies better for beginners that who have decent understanding of divide and conquer. However, once you study further and have enough practice, the tabulation should be more intuitive compared with recursive solution. Usually, dynamic programming solution to a problem refers to the solution with tabulation.

We enumerate two examples: Fibonacci Sequence (Subsection 16.1.3) and Longest Increasing Subsequence (subsection ??) in the remaining section to showcase *memoization* and *tabulation* in practice.

16.1.3 Fibonacci Sequence

Problem Definition Given $f(0) = 0, f(1) = 1, f(n) = f(n - 1) + f(n - 2), n \geq 2$. Return the value for any given n .

As the most elementary and classical example demonstrating dynamic programming, we carry on this tradition and give multi-fold of solutions for fibonacci sequence. Since in Chapter. 13 the recursive and naive solution is already given, we will just briefly explain it here.

Complete Search Because the relation between current state and previous states are directly given, it is straightforward to solve the problem in

a top-down fashion using depth-first search. The time complexity can be easily obtained from using induction or recursion tree: $O(2^n)$, where the base 2 is the width of the tree, and n is the depth. The Python code is given:

```

1 # DFS on subproblem graph
2 def fibonacciDFS(n):
3     # base case
4     if n <= 1: return n
5     return fibonacciDFS(n-1)+fibonacciDFS(n-2) # use the result
       of subtree to build up the result of current tree.
```

Memoization As we explained, there are subproblems computed more than once in the complete search solution. To avoid the recomputation, we can use a hashtable `memo` to save the solved subproblem. We need to make `memo` globally and available for all recursive calls; in the memoized complete search, instead of calling the recursion function $f(n) = f(n-1) + f(n-2)$ to get answer for current state n , it first check if the problem is already solved and available in `memo`.

Because to solve $f(n)$, there will be n subproblems, and each subproblem only depends on two smaller problems, so the time complexity will be lowered to $O(n)$ if we use DFS+memoizataion.

```

1 # DFS on subproblem graph + Memoization
2 def fibonacciDFSMemo(n, memo):
3     if n <= 1: return n
4     if n not in memo:
5         memo[n] = fibonacciDFSMemo(n-1, memo)+fibonacciDFSMemo(n
6             -2, memo)
6     return memo[n]
```

Bottom-up Tabulation In the top-down recursive solution, where exists two passes: one pass to divide the problems into subproblems, and the other recursive pass to gather solution from the base case and construct solution for larger problems. However, in the bottom-up tabulation way, for this specific problem, we have four key steps:

1. We start by *assigning* `dp` array to save each state's result. It represents the fibonacci number at each index (from 0 to n), which is also called a *state*.
2. Then, we *initialize* results of base cases which either were given or can be obtained easily with simple deduction.
3. We iterate through each subproblem/state in reversed topological sort order, which is $[0, 1, 2, 3, 4]$ as in Fig 16.2, and use tabulazied solution to build up the answer of current state through the given *recurrence function* $f(n) = f(n - 1) + f(n - 2)$.

4. We return the last state in `dp` as the final *answer*.

The tabulation code of dynamic programming is given:

```

1 # Dynamic Programming: bottom-up tabulation O(n) , O(n)
2 def fibonacciDP(n):
3     dp = [0]*(n+1)
4     # init
5     dp[1] = 1
6     for i in range(2,n+1):
7         dp[i] = dp[i-1] + dp[i-2]
8     return dp[n]
```

16.2 Dynamic Programming Knowledge Base

So far, we have learned most of the knowledge related to Dynamic programming, including basic concepts and two examples. In this section, we would officially answer three questions – *when* and *how* to apply dynamic programming? and *which* type of dynamic programming we need? Tabulation or Memoization. With clear definition, and offering some more practical guideline, complexity analysis, and comprehensive comprehension between memoization and tabulation, we are determined to demystify dynamic programming. The subsections are organized as:

1. Two properties and Practical Guideline (Section 16.2.1) that an optimization problems must have in order to answer the *when* question.
2. Five key elements, General Steps to Solve Dynamic Programming, and Complexity Analysis (Section 16.2.2) in implementing the dynamic programming solution and to answer the *how* question.
3. Tabulation VS Memoization (Section 16.2.3) to answer the *which* question.

16.2.1 When? Two properties

In order for the dynamic programming to apply, these two properties: overlapping subproblems and optimal substructure must be found in our solving problems. From our illustrated examples, 1) the step of identifying overlapping shows the overlapping subproblem properties. 2) the recurrence function in fact shows the optimal substructure. To be official, these two essential properties states as:

Overlapping Subproblems When a recursive algorithm revisits the same subproblem repeatedly, we say that the optimization problem has overlapping subproblems. This can be easily visualized in the top-down subproblem graph, where one state is reached by multiple other states. This property

demonstrates the recomputation overhead seen in the complete search solutions of our two examples.

Overlapping Subproblems property helps us find space for optimization and lead us to its solution – *the caching mechanism* used in dynamic programming. In the flip side, when subproblems are disjoint such as seen in merge sort and binary search, dynamic programming would not be helping.

Optimal Substructure A given problem has optimal substructure property if the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems. Only if optimal substructure property applied we can find the *recurrence relation function* which is a key step in implementation as we have seen from the above two examples. Optimal substructures varies across problem domains in two ways:

1. **Subproblem space:** how many subproblems an optimal solution to the original problem uses. For example, in Fibonacci sequence, each integer in range $[0, n]$ is a subproblem, which makes the $n + 1$ as the total subproblem space. The state of each subproblem is the optimal solution for that subproblem which is $f(n)$. And in the example of LIS, each subproblem is the array with index in range $[0, i]$, and its state is the length of the longest increasing subsuquence ends/includes at index i , this makes the whole subproblem space to be n too.
2. **State Choice:** how many choices we have in determining which subproblem(s) to use to decide the recurrence function for the current state. In Fibonacci sequence, each state only relies on two preceding states as seen in recurrence function $f(i) = f(i - 1) + f(i - 2)$, thus making it constant cost. For LIS, each state require knowing all inclusive states (solutions relating to all smaller subproblems), which makes it cost of $O(n)$ that relates to the subproblem space.

Subproblem space and state choice together not only formulates the recurrent relation with which we very much have the implementation in hand. Together they also decide the time and space complexity we will need to tackle our dynamic programming problems.

Practical Guideline Instead of the textbook definition, we also summarize the experience shared by experienced software programmers. Dynamic programming problems are normally asked in its certain way and its naive solution shows certain time complexity. Here, we summarize the situations when to use or not to use dynamic programming as **Dos** and **Donots**.

- **Dos:** Dynamic programming fits for the optimizing the following problems which are either exponential or polynomial complexity using complete search:

1. Optimization: Compute the maximum or minimum value;
 2. Counting: Count the total number of solutions;
 3. Checking if a solution works.
- **Donots:** In the following cases we might not be able to apply Dynamic Programming:
 1. When the naive solution has a low time complexity already such as $O(n^2)$ or $O(n^3)$.
 2. When the input dataset is a set while not an array or string or matrix, 90% chance we will not use DP.
 3. When the overlapping subproblems apply but it is not optimization problems thus we can not identify the suboptimal substructure property and thus find its recurrence function. For example, same problem context as in **Dos** but instead we are required to obtain or print all solutions, this is when we need to retreat back to the use of DFS+memo(top-down) instead of DP.

16.2.2 How? Five Elements and Steps

In Section, we have provided two forms of dynamic programming solutions: memoization and tabulation, as two different ways of bringing the caching mechanism into practice. In this section, we focus on the iterative tabulation and generalize its four key elements and practical guidelines for import steps.

Five Key Elements of Tabulation As the first guideline of Tabulation, we summarize the four key elements for the implementation of dynamic programming:

1. **Subproblem and State:** Define what the subproblem space, what is the optimal state/solution for each subproblem. In practice, it would normally be the *the total/the maximum/minimum* for subproblem. This requires us to know how to divide problem into subproblems, there are patterns to follow which will be detailed in Chapter. ??.
2. **State Transfer (Recurrence) Function:** derive the function that how we can get current state by using result from previous computed state(s). This requires us to identify the optimal substructure and know how to make state choice.
3. **Assignment and Initialization:** Followed by knowing the subproblem space, we typically assign a space data structure and initialize its values. For base or edge cases, we might need to initialize different than the other more general cases.

4. **Iteration:** decide the order of iterating through the subproblem space thus we can scan each subproblem/state exact and only *once*. Using the subproblem graph, and visit the subproblems in reversed topological order is a good way to go.
5. **Answer:** decide which state or a combination of all states such as the the max/min of all the state is the final result needed.

Five Steps to Solve Dynamic Programming This is a general guideline for dynamic programming – memoization or tabulation. Key advice – being “flexible”. Given a real problem, all in all, we are credited with our understanding of the concepts in computer science. Thus, we should not be too bothered or stressed that if you can not come up with a “perfect” answer.

1. Read the question: search for the key words of the problem patterns: counting, checking, or maximum/minimum.
2. Come up with the most naive solution ASAP: analyze its time complexity. Is it a typical DFS solution? Try draw a SUBPROBLEM GRAPH to get visualization. Is there space for optimization?
3. Apply Section 16.2.1: Is there overlapping? Can you define the optimal substructure/recurrence function?
4. If the conclusion is YES, try to define the Five key elements so that we can solve it using the preferable tabulation. If you can figure it out intuitively just like that, great! What to do if not? Maybe retreat to use memoization, which is a combination of divide and conquer, DFS, and memoization.
5. What if we were just so nervous that or time is short, we just go ahead and implement the complete search solution instead. With implementation is better than nothing. With the implementation in hand, maybe we can figure it out later.

Complexity Analysis The complexity analysis of the tabulation is seemingly more straightforward compared with its counterpart – the recursive memoization. For the tabulation, we can simply draw conclusion without any prior knowledge of the dynamic programming by observing the `for` loops and its recurrence function. However, for both variant, there exists a common analysis method. The core points to analyze complexity involving dynamic programming is: (1) the subproblem space $|S|$, that is the total number of subproblems; and (2) the number of state choice needed to construct each state $|C|$. By multiplying these two points, we can draw the conclusion of its time complexity as $O(|S||C|)$.

For example, if the subproblem space is n and if each state i relies on (1) only one or two previous states as we have seen in the example of Fibonacci Sequence, it makes the time complexity $O(n)$; and (2) all previous states in range $[0, i - 1]$ as seen in the example of Longest Increasing Subsequence, which can be viewed as $O(n)$ to solve each subproblem, this brings up the complexity up to $O(n^2)$.

16.2.3 Which? Tabulation or Memoization

As we can see, the way the bottom-up DP table is filled is not as intuitive as the top-down DP as it requires some ‘reversals’ of the signs in Complete Search recurrence that we have developed in previous sections. However, we are aware that some programmers actually feel that the bottom-up version is more intuitive. The decision on using which DP style is in your hand. To help you decide which style that you should take when presented with a DP solution, we present the trade-off comparison between top-down Memoization and bottom-up Tabulation in Table 16.1.

Table 16.1: Tabulation VS Memoization

	Memoization	Tabulation
Pros	<ul style="list-style-type: none"> 1. A natural transformation from normal recursive complete search. 2. Compute subproblems only when necessary, sometimes this can be faster. 	<ul style="list-style-type: none"> 1. Faster if many sub-problems are revisited as there is no overhead of recursive calls. 2. Can save memory space with dynamic programming ‘on-the-fly’ technique (see Section Extension ??).
Cons	<ul style="list-style-type: none"> 1. Slower if many subproblems are revisited due to overhead of recursive calls. 2. If there are n states, it can use up to $O(n)$ table size which might lead to Memory Limit Exceeded(MLE) for some hard problems. 3. Faces stack overflow due to the recursive calls. 	<ul style="list-style-type: none"> 1. For programmers who are inclined with recursion, this may not be intuitive.

16.3 Hands-on Examples (Main-course Examples)

In the practical guideline, we mentioned that the problems that can be further optimized with dynamic programming would be seen with complexity

patters of their naive solutions: either exponential such as $O(2^n)$ or polynomial such as $O(n^3)$ or $O(n^2)$.

The purpose of this section is to further enhance our knowledge and put our both theoretical and practical guideline into test. We examine two examples: Triangle and maximum subarray. We have seen how maximum subarray can be solved with linear search and divide and conquer in Chapter.???. However, in this section, we expand the old solution into dynamic programming solutions and we see the difference and connection.

16.3.1 Exponential Problem: Triangle

Triangle (L120) Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

Example :

Given the following triangle :

```
[
[2] ,
[3,4] ,
[6,5,7] ,
[4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., $2 + 3 + 5 + 1 = 11$).

Analysis

1. We quickly read the question and we find the key word – minimum.
2. We come up with the most naive solution that would be dfs which we have already covered in chapter. A quick drawing of dfs traversal graph, we can find some nodes are repetitively visited.
3. Apply Two Properties: First, define the subproblem, for each node in the triangle, it is decided by two indexes (i, j) as row and column index respectively. The subproblem can be straightforward, the minimum sum from the starting point $(0, 0)$ to current position (i, j) . And the subprogram graph will be exactly the same as the graph we used in dfs. We identify overlapping easily.

Now, develop the recurrence function. To build up solution for state at (i, j) , it needs two other states: $(i - 1, j)$ and $(i - 1, j - 1)$ and need one value from current state. The function will be: $f(i, j) = \min(f(i - 1, j), f(i - 1, j - 1)) + t[i][j]$.

4. Five Key Elements: we need to figure out how to assign and initialize the **dp** space and do the iteration. To get the boundary condition:

- (a) by observation: the first element at $(0, 0)$ will have none of these two states $f(i - 1, j), f(i - 1, j - 1)$ exist. the leftmost and rightmost element of the triangle will have only one of these two states: $f(i - 1, j), f(i - 1, j - 1)$.
- (b) by simple math induction: $i \in [0, n - 1], j \in [0, i]$. When $i = 0, j = 0$, $f(i, j) = t[i][j]$, when $i \in [1, n - 1], j = 0$, $f(i - 1, j - 1)$ is invalid, and when $i = n - 1, j = n - 1$, $(i - 1, j)$ is invalid.

The answer would be the minimum value of dp at the last row. The Python code is given:

```

1 def min_path_sum(t):
2     dp = [[0 for c in range(r+1)] for r in range(len(
3         triangle))] # initialized to 0 for f()
4     n = len(triangle)
5     #initialize the first point, bottom
6     dp[0][0] = triangle[0][0]
7     #initial the left col and the right col of the triangle
8     for i in range(1, n):
9         dp[i][0] = dp[i-1][0] + dp[i][0]
10        dp[i][i] = dp[i-1][i-1] + dp[i][i]
11        for i in range(1, n):
12            for j in range(1, i):
13                dp[i][j] = t[i][j] + min(dp[i-1][j], dp[i-1][j
-1])
14    return min(dp[-1])

```

Space Optimization From the recurrence function, we can see the current state is only related to two states from the last row. We can reuse the original `triangle` matrix itself to save the state. If we are following the forward induction as the previous solution, we still have the problem of edge cases; for some state that it only has one previous or none previous states needed to decide its current state. We can write our code as:

```

1 def min_path_sum(t):
2     """
3     Space optimization with forward induction
4     """
5     t = deepcopy(t)
6     if not t:
7         return 0
8     n = len(t)
9     for i in range(0, n):
10        for j in range(0, i + 1):
11            if i == 0 and j == 0:
12                continue
13            elif j == 0:
14                t[i][j] = t[i][j] + t[i-1][j]
15            elif j == i:
16                t[i][j] = t[i][j] + t[i-1][j-1]
17            else:

```

```

18     t[i][j] = t[i][j] + min(t[i-1][j], t[i-1][j-1])
19     return min(t[-1])

```

Further Optimization Let us look at the traversal order backward where we start from the last row and traverse upward to the first row. For the last row, its state should be the same as its triangle value. For any remaining rows and each of its element, its state will all rely on two other states located below of them. There is consistency in this backward induction and the final state at the first row will be only final global answer. In this method, we reverse of recurrence function as $f(i, j) = \min(f(i+1, j+1), f(i+1, j)) + t[i][j]$.

```

1 def min_path_sum(t):
2     """
3     Space optimization with backward induction
4     """
5     t = deepcopy(t)
6     if not t:
7         return 0
8     n = len(t)
9     # Start from the last second row
10    for i in range(n-2, -1, -1):
11        for j in range(i, -1, -1):
12            t[i][j] = t[i][j] + min(t[i+1][j], t[i+1][j+1])
13    return t[0][0]

```

16.3.2 Polynomial Problem: Maximum Subarray

Maximum Subarray (L53) Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$, the contiguous subarray $[4, -1, 2, 1]$ has the largest sum = 6.

The problem will be analyzed following our two properties and solved following our five step guideline and five elements.

Analysis and $O(n)$ Solution

1. First step, we read the problem and we can quickly catch the key word – maximum.
2. Second step, the naive solution. We have From other chapters, we have seen how maximum subarray can be approached as either graph search ($O(2^n)$ to get more details later), linear search along the solution space ($O(n^3)$ and $O(n^2)$ if be tweeted with the computation of subarray).
3. Third step: Apply two properties. The solution space we concluded for maximum subarray would be totally in $O(n^2)$, and be denoted as

$a[i, j]$ where $i, j \in [0, n - 1], j \leq i$. This states that the maximum subarray is one of these subarrays fixing their starting index. Here, in order to think in dynamic programming way, let us define subproblem. We first define it as $a[i, j]$, and the state would be its sum of this subarray. We can see there is already some hidden recurrence function that $f(i, j) = f(i, j - 1) + a[j]$. We would see there is overlap: $a[0, 4]$ actually includes $a[1, 4]$.

However, there is something missing. The state we define did not take leverage of the optimal substructure. Let us define the subproblem in another way that has the optimal condition there. We define $f(i), i \in [0, n - 1]$, represents the subarry that starts from index i and the answer/state will be the maximum value of these potential subarrays. Therefore, the subproblem space will be only $O(n)$. The solution space of subproblem $f(i)$ is $a[i, j]$ where $i, j \in [0, n - 1], j \leq i$. Assume we are comparing $f(0)$ and $f(1)$, that is the relation of maximum subarry starts from 0 and the maximum subarry that starts from 1. $f(1)$ is a subproblem of $f(0)$. If $f(1)$ is computed already, the $f(0)$ would be either include $f(1)$ if its positive or not include with two possible state-choice. Thus, we get our recurrence function $f(i - 1) = \max(f(i) + a[i], a[i])$. The last state is $f(n - 1) = a[n - 1]$.

4. Step 4: Given all the conclusions, we can start the five key elements. The above solution requires us to start from the maximum index in a reverse order, this is called *backward induction* mentioned in materials explaining dynamic programming from the angle of optimization. We need to always pay attention there is empty array where the maximum subarray should give zero as result. This makes our total states $n + 1$ instead of n . In the backward induction, this empty state will locate at index n with a list of size $n + 1$.

```

1 def maximum_subarray_dp(a):
2     """
3     Backward induction dp solution
4     """
5     # assignment and initialization
6     dp = [0] * (len(a) + 1)
7     # fill out the dp space in reverse order
8     # we do not need to fill the base case dp[n]
9     for i in reversed(range(len(a))):
10         dp[i] = max(dp[i+1] + a[i], a[i])
11     print(dp)
12     return max(dp)

```

Space Optimization If we observe the iterating process, we always only use one previous state. If we use another global variable, say `maxsum` to track

the global maximum subarray value, and use `state` to replace `dp` array, we can decrease the space complexity from $O(n)$ to $O(1)$.

```

1 def maximum_subarray_dp_sp(a):
2     """
3     dp solution with space optimization
4     """
5     # assignment and initialization
6     state = 0
7     maxsum = 0
8     # fill out the dp space in reverse order
9     # we do not need to fill the base case dp[n]
10    for i in reversed(range(len(a))):
11        state = max(state + a[i], a[i])
12        maxsum = max(maxsum, state)
13    return maxsum

```

All of the above steps are for deep analysis purpose. When you are more experienced, we can go directly to the five elements of tabulation and develop the solution without connecting it to the naive solution. Also, this is actually a Kadane's Algorithm which will be further detailed in Chapter. ??.

16.4 Exercises

16.4.1 Knowledge Check

1. The completeness of Dynamic programming.

16.4.2 Coding Practice

In order to understand how the efficiency is boosted from searching algorithms to dynamic programming, readers will be asked to give solutions for both searching algorithms and dynamic programming algorithms. And then to compare and analyze the difference. (Two problems to be asked)

1. **Coordinate Type 63. Unique Paths II** (medium).

A robot is located at the top-left corner of a $m \times n$ grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

Note: m and n will be at most 100.

Example 1:

```

1 Input:
2 [
3   [0 ,0 ,0] ,
4   [0 ,1 ,0] ,
5   [0 ,0 ,0]
6 ]
7 Output: 2

```

Explanation: There is one obstacle in the middle of the 3x3 grid above. There are two ways to reach the bottom-right corner:

1. Right → Right → Down → Down
2. Down → Down → Right → Right

Sequence Type

2. 213. House Robber II

Note: This is an extension of House Robber.

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

example nums = [3,6,4], return 6

```

1 def rob( self ,  nums):
2     """
3         :type  nums:  List [ int ]
4         :rtype:  int
5     """
6
7     if  not  nums:
8         return 0
9     if  len( nums)==1:
10        return  nums[ 0]
11    def  robber1( nums):
12        dp=[ 0]* ( 2)
13        dp[ 0] =0
14        dp[ 1] =nums[ 0] #if  len  is  1
15        for  i  in  range( 2, len( nums)+1): #if  leng  is
16            2....,  index  is  i-1
17            dp[ i%2]=max( dp[ ( i-2)%2]+nums[ i-1],  dp[ ( i-1)
18            %2])
19        return  dp[ len( nums)%2]
20
21    return  max( robber1( nums[ : -1]), robber1( nums[ 1: ]))

```

3. 337. House Robber III

4. 256. Paint House

There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a $n \times 3$ cost matrix. For example, $\text{costs}[0][0]$ is the cost of painting house 0 with color red; $\text{costs}[1][2]$ is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Solution: state: 0, 1, 2 colors $\text{minCost}[i] = \text{till } i$ the mincost for each color for color 0: paint 0 [0] = $\min(\text{minCost}[i-1][1], \text{minCost}[i-1][2]) + \text{costs}[i][0]$

paint 1 [1]

$\text{minCost}[i] = [0,1,2]$, i for i in [0,1,2]

answer = $\min(\text{minCost}[-1])$

```

1 def minCost(self, costs):
2     """
3         :type costs: List[List[int]]
4         :rtype: int
5     """
6     if not costs:
7         return 0
8     if len(costs) == 1:
9         return min(costs[0])
10
11    minCost = [[0 for col in range(3)] for row in range
12    (len(costs)+1)]
13    minCost[0] = [0, 0, 0]
14    minCost[1] = [cost for cost in costs[0]]
15    colorSet=set([1, 2, 0])
16    for i in range(2, len(costs)+1):
17        for c in range(3):
18            #previous color
19            pres = list(colorSet-set([c]))
20            print(pres)
21            minCost[i][c] = min([minCost[i-1][pre_cor]
for pre_cor in pres])+costs[i-1][c]
            return min(minCost[-1])

```

5. 265. Paint House II

There are a row of n houses, each house can be painted with one of the k colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a $n \times k$ cost matrix. For example, $\text{costs}[0][0]$ is the cost of painting house 0 with color 0; $\text{costs}[1][2]$ is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

Note: All costs are positive integers.

Follow up: Could you solve it in $O(nk)$ runtime?

Solution: this is exactly the same as the last one:

```

1 if not costs:
2     return 0
3     if len(costs)==1:
4         return min(costs[0])
5
6     k = len(costs[0])
7     minCost = [[0 for col in range(k)] for row in range
8     (len(costs)+1)]
9     minCost[0] = [0]*k
10    minCost[1]=[cost for cost in costs[0]]
11    colorSet=set([i for i in range(k)])
12    for i in range(2,len(costs)+1):
13        for c in range(k):
14            #previous color
15            pres = list(colorSet-set([c]))
16            minCost[i][c] = min([minCost[i-1][pre_cor]
for pre_cor in pres])+costs[i-1][c]
return min(minCost[-1])

```

6. 276. Paint Fence

There is a fence with n posts, each post can be painted with one of the k colors.

You have to paint all the posts such that no more than two adjacent fence posts have the same color.

Return the total number of ways you can paint the fence.

Note: n and k are non-negative integers. for three posts, the same color, the first two need to be different

```

1 def numWays(self , n , k):
2     """
3     :type n: int
4     :type k: int
5     :rtype: int
6     """
7     if n==0 or k==0:
8         return 0
9     if n==1:
10        return k
11
12     count = [[0 for col in range(k)] for row in range(n
+1)]

```

```

13     same = k
14     diff = k*(k-1)
15     for i in range(3,n+1):
16         pre_diff = diff
17         diff = (same+diff)*(k-1)
18         same = pre_diff
19     return (same+diff)

```

Double Sequence Type DP

7. 115. Distinct Subsequences (hard)

Given a string S and a string T, count the number of distinct subsequences of S which equals T.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Example 1:

```

1 Input: S = "rabbbit" , T = "rabbit"
2 Output: 3

```

Explanation:

As shown below, there are 3 ways you can generate "rabbit" from S. (The caret symbol $\hat{}$ means the chosen letters)

```

1 rabbbit
2 ^~~~~ ~
3 rabbbit
4 ^~ ~~~
5 rabbbit
6 ^~~ ~~~

```

8. 97. Interleaving String

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

Example 1:

```

1 Input: s1 = "aabcc" , s2 = "dbbca" , s3 = "aadbbebcac"
2 Output: true

```

Example 2:

```

1 Input: s1 = "aabcc" , s2 = "dbbca" , s3 = "aadbbebaccc"
2 Output: false

```

Splitting Type DP

9. 132. Palindrome Partitioning II (hard)

Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

Example:

```
1 Input: "aab"
2 Output: 1
```

Explanation: The palindrome partitioning ["aa","b"] could be produced using 1 cut.

Exercise: max difference between two subarrays: An integer indicate the value of maximum difference between two Subarrays. The temp java code is:

```
1 public int maxDiffSubArrays(int[] nums) {
2     // write your code here
3     int size = nums.length;
4     int[] left_max = new int[size];
5     int[] left_min = new int[size];
6     int[] right_max = new int[size];
7     int[] right_min = new int[size];
8
9     int localMax = nums[0];
10    int localMin = nums[0];
11
12    left_max[0] = left_min[0] = nums[0];
13    //search for left_max
14    for (int i = 1; i < size; i++) {
15        localMax = Math.max(nums[i], localMax + nums[i]);
16        left_max[i] = Math.max(left_max[i - 1], localMax);
17    }
18    //search for left_min
19    for (int i = 1; i < size; i++) {
20        localMin = Math.min(nums[i], localMin + nums[i]);
21        left_min[i] = Math.min(left_min[i - 1], localMin);
22    }
23
24    right_max[size - 1] = right_min[size - 1] = nums[size - 1];
25    //search for right_max
26    localMax = nums[size - 1];
27    for (int i = size - 2; i >= 0; i--) {
28        localMax = Math.max(nums[i], localMax + nums[i]);
29        right_max[i] = Math.max(right_max[i + 1], localMax);
```

```

30
31     }
32     //search for right min
33     localMin = nums[ size - 1];
34     for ( int i = size - 2; i >= 0; i-- ) {
35         localMin = Math.min( nums[ i ] , localMin + nums[ i
36             ] );
37         right_min[ i ] = Math.min( right_min[ i + 1 ] ,
38             localMin );
39         }
40         //search for separate position
41         int diff = 0;
42         for ( int i = 0; i < size - 1; i++ ) {
43             diff = Math.max( Math.abs( left_max[ i ] -
44                 right_min[ i + 1 ] ) , diff );
45             diff = Math.max( Math.abs( left_min[ i ] -
46                 right_max[ i + 1 ] ) , diff );
47         }
48         return diff;
49     }

```

10. 152. Maximum Product Subarray (medium)

Given an integer array `nums`, find the contiguous subarray within an array (containing at least one number) which has the largest product.

Example 1:

```

1 Input: [2,3,-2,4]
2 Output: 6
3 Explanation: [2,3] has the largest product 6.

```

Example 2:

```

1 Input: [-2,0,-1]
2 Output: 0
3 Explanation: The result cannot be 2, because [-2,-1] is not
   a subarray.

```

Solution: this is similar to the maximum sum subarray, the difference we need to have two local vectors, one to track the minimum value: `min_local`, the other is `max_local`, which denotes the minimum and the maximum subarray value including the i th element. The function is as follows.

$$\min_{local}[i] = \begin{cases} \min(\min_{local}[i-1] * \text{nums}[i], \text{nums}[i]), & \text{nums}[i] < 0; \\ \min(\max_{local}[i-1] * \text{nums}[i], \text{nums}[i]) & \text{otherwise} \end{cases} \quad (16.1)$$

$$\max_{local}[i] = \begin{cases} \max(\max_{local}[i-1] * \text{nums}[i], \text{nums}[i]), & \text{nums}[i] > 0; \\ \max(\min_{local}[i-1] * \text{nums}[i], \text{nums}[i]) & \text{otherwise} \end{cases} \quad (16.2)$$

```

1 def maxProduct(nums):
2     if not nums:
3         return 0
4     n = len(nums)
5     min_local, max_local = [0]*n, [0]*n
6     max_so_far = nums[0]
7     min_local[0], max_local[0] = nums[0], nums[0]
8     for i in range(1, n):
9         if nums[i]>0:
10            max_local[i] = max(max_local[i-1]*nums[i], nums[i])
11            min_local[i] = min(min_local[i-1]*nums[i], nums[i])
12        else:
13            max_local[i] = max(min_local[i-1]*nums[i], nums[i])
14            min_local[i] = min(max_local[i-1]*nums[i], nums[i])
15        max_so_far = max(max_so_far, max_local[i])
16    return max_so_far

```

With space optimization:

```

1 def maxProduct(self, nums):
2     if not nums:
3         return 0
4     n = len(nums)
5     max_so_far = nums[0]
6     min_local, max_local = nums[0], nums[0]
7     for i in range(1, n):
8         if nums[i]>0:
9             max_local = max(max_local*nums[i], nums[i])
10            min_local = min(min_local*nums[i], nums[i])
11        else:
12            pre_max = max_local #save the index
13            max_local = max(min_local*nums[i], nums[i])
14            min_local = min(pre_max*nums[i], nums[i])
15        max_so_far = max(max_so_far, max_local)
16    return max_so_far

```

Even simpler way to write it:

```

1 def maxProduct(self, nums):
2     if not nums:
3         return 0
4     n = len(nums)
5     max_so_far = nums[0]
6     min_local, max_local = nums[0], nums[0]
7     for i in range(1, n):
8         a = min_local*nums[i]
9         b = max_local*nums[i]
10        max_local = max(nums[i], a, b)
11        min_local = min(nums[i], a, b)
12        max_so_far = max(max_so_far, max_local)
13    return max_so_far

```

11. 122. Best Time to Buy and Sell Stock II

Say you have an array for which the i th element is the price of a given stock on day i .

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

```

1 Input: [7,1,5,3,6,4]
2 Output: 7
3 Explanation: Buy on day 2 (price = 1) and sell on day 3 (
               price = 5), profit = 5-1 = 4.
4                 Then buy on day 4 (price = 3) and sell on day
5 (price = 6), profit = 6-3 = 3.
```

Example 2:

```

1 Input: [1,2,3,4,5]
2 Output: 4
3 Explanation: Buy on day 1 (price = 1) and sell on day 5 (
               price = 5), profit = 5-1 = 4.
4 Note that you cannot buy on day 1, buy on day
5 2 and sell them later, as you are
   engaging multiple transactions at the same
   time. You must sell before buying again.
```

Example 3:

```

1 Input: [7,6,4,3,1]
2 Output: 0
3 Explanation: In this case, no transaction is done, i.e. max
               profit = 0.
```

Solution: the difference compared with the first problem is that we can have multiple transaction, so whenever we can make profit we can have an transaction. We can notice that if we have [1,2,3,5], we only need one transaction to buy at 1 and sell at 5, which makes profit 4. This problem can be resolved with decreasing monotonic stack. whenever the stack is increasing, we kick out that number, which is the smallest number so far before i and this is the transaction that make the biggest profit = current price - previous element. Or else, we keep push smaller price inside the stack.

```

1 def maxProfit(self, prices):
2     """
3         :type prices: List[int]
4         :rtype: int
5     """
```

```

6     mono_stack = []
7     profit = 0
8     for p in prices:
9         if not mono_stack:
10            mono_stack.append(p)
11        else:
12            if p<mono_stack[-1]:
13                mono_stack.append(p)
14            else:
15                #kick out till it is decreasing
16                if mono_stack and mono_stack[-1]<p:
17                    price = mono_stack.pop()
18                    profit += p-price
19
20                while mono_stack and mono_stack[-1]<p:
21                    price = mono_stack.pop()
22                    mono_stack.append(p)
23    return profit

```

12. 188. Best Time to Buy and Sell Stock IV (hard)

Say you have an array for which the i th element is the price of a given stock on day i .

Design an algorithm to find the maximum profit. You may complete at most k transactions.

Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Example 1:

```

1 Input: [2,4,1], k = 2
2 Output: 2
3 Explanation: Buy on day 1 (price = 2) and sell on day 2 (
               price = 4), profit = 4-2 = 2.

```

Example 2:

```

1 Input: [3,2,6,5,0,3], k = 2
2 Output: 7
3 Explanation: Buy on day 2 (price = 2) and sell on day 3 (
               price = 6), profit = 6-2 = 4.
4           Then buy on day 5 (price = 0) and sell on day
6           (price = 3), profit = 3-0 = 3.

```

13. 644. Maximum Average Subarray II (hard)

Given an array consisting of n integers, find the contiguous subarray whose length is greater than or equal to k that has the maximum average value. And you need to output the maximum average value.

Example 1:

```

1 Input: [1,12,-5,-6,50,3], k = 4
2 Output: 12.75
3 Explanation:
4 when length is 5, maximum average value is 10.8,
5 when length is 6, maximum average value is 9.16667.
6 Thus return 12.75.

```

Note:

```

1 1 <= k <= n <= 10,000.
2 Elements of the given array will be in range [-10,000,
10,000].
3 The answer with the calculation error less than 10^-5
will be accepted.

```

14. Backpack Type Backpack II Problem

Given n items with size $A[i]$ and value $V[i]$, and a backpack with size m . What's the maximum value can you put into the backpack? Notice You cannot divide item into small pieces and the total size of items you choose should smaller or equal to m . Example

```

1 Given 4 items with size [2, 3, 5, 7] and value [1, 5, 2,
4], and a backpack with size 10. The maximum value is 9.

```

Challenge

$O(n \times m)$ memory is acceptable, can you do it in $O(m)$ memory? Note Hint: Similar to the backpack I, difference is $dp[j]$ we want the value maximum, not to maximize the volume. So we just replace $f[i-A[i]]+A[i]$ with $f[i-A[i]]+V[i]$.

15. 801. Minimum Swaps To Make Sequences Increasing

```

1 We have two integer sequences A and B of the same non-zero
length .
2
3 We are allowed to swap elements A[i] and B[i]. Note that
both elements are in the same index position in their
respective sequences .
4
5 At the end of some number of swaps , A and B are both
strictly increasing . (A sequence is strictly increasing
if and only if  $A[0] < A[1] < A[2] < \dots < A[A.length - 1]$  .)
6
7 Given A and B, return the minimum number of swaps to make
both sequences strictly increasing . It is guaranteed
that the given input always makes it possible .
8
9 Example :
10 Input: A = [1,3,5,4], B = [1,2,3,7]
11 Output: 1

```

```

12 Explanation:
13 Swap A[3] and B[3]. Then the sequences are:
14 A = [1, 3, 5, 7] and B = [1, 2, 3, 4]
15 which are both strictly increasing.
16
17 Note:
18
19 A, B are arrays with the same length, and that length
20 will be in the range [1, 1000].
A[i], B[i] are integer values in the range [0, 2000].

```

Simple DFS. The brute force solution is to generate all the valid sequence and find the minimum swaps needed. Because each element can either be swapped or not, thus make the time complexity $O(2^n)$. If we need to swap current index i is only dependent on four elements at two state, $(A[i], B[i], A[i-1], B[i-1])$, at state i and $i-1$ respectively. At first, supposedly for each path, we keep the last visited element a and b for element picked for A and B respectively. Then

```

1 def minSwap(self, A, B):
2     if not A or not B:
3         return 0
4
5     def dfs(a, b, i): #the last element of the state
6         if i == len(A):
7             return 0
8         if i == 0:
9             # not swap
10            count = min(dfs(A[i], B[i], i+1), dfs(B[i], A[i],
11                i+1)+1)
12            return count
13        count = sys.maxsize
14
15        if A[i]>a and B[i]>b: #not swap
16            count = min(dfs(A[i], B[i], i+1), count)
17        if A[i]>b and B[i]>a:#swap
18            count = min(dfs(B[i], A[i], i+1)+1, count)
19
20    return dfs([], [], 0)

```

DFS with single State Memo is not working. Now, to avoid overlapping, [5,4], [3,7] because for the DFS there subproblem is in reversed order compared with normal dynamic programming. Simply using the index to identify the state will not work and end up with wrong answer.

DFS with multiple choiced memo. For this problem, it has two potential choice, swap or keep. The right way is to distinguish different state with additional variable. Here we use *swapped* to represent if the current level we make the decision of swap or not.

```

1 def minSwap(self, A, B):
2     if not A or not B:
3         return 0
4
5     def dfs(a, b, i, memo, swapped): #the last element of
6         the state
7             if i == len(A):
8                 return 0
9             if (swapped, i) not in memo:
10                 if i == 0:
11                     # not swap
12                     memo[(swapped, i)] = min(dfs(A[i], B[i], i
13 +1, memo, False), dfs(B[i], A[i], i+1, memo, True)+1)
14                 return memo[(swapped, i)]
15             count = sys.maxsize
16
17             if A[i]>a and B[i]>b: #not swap
18                 count = min(count, dfs(A[i], B[i], i+1,
19 memo, False))
20                 if A[i]>b and B[i]>a: #swap
21                     count = min(count, dfs(B[i], A[i], i+1,
22 memo, True) +1)
23                     memo[(swapped, i)] = count
24
25             return memo[(swapped, i)]
26
27     return dfs([], [], 0, {}, False)

```

Dynamic Programming. Because it has two choice, we define two dp state arrays. One represents the minimum swaps if current i is not swapped, and the other is when the current i is swapped.

```

1 def minSwap(self, A, B):
2     if not A or not B:
3         return 0
4
5     dp_not =[sys.maxsize]*len(A)
6     dp_swap = [sys.maxsize]*len(A)
7     dp_swap[0] = 1
8     dp_not[0] = 0
9     for i in range(1, len(A)):
10         if A[i] > A[i-1] and B[i] > B[i-1]: #i-1 not swap
11             and i not swap
12                 dp_not[i] = min(dp_not[i], dp_not[i-1])
13                 # if i-1 swap, it means A[i]>B[i-1], i need to
14                 swap
15                 dp_swap[i] = min(dp_swap[i], dp_swap[i-1]+1)
16                 if A[i] > B[i-1] and B[i] > A[i-1]: # i-1 not swap,
17                 i swap
18                     dp_swap[i] = min(dp_swap[i], dp_not[i-1]+1)
19                     # if i-1 swap, it means the first case, current
20                     need to not to swap
21                     dp_not[i] = min(dp_not[i], dp_swap[i-1])
22     return min(dp_not[-1], dp_swap[-1])

```

Actually, in this problem, the DFS+memo solution is not easy to understand any more. On the other hand, the dynamic programming is easier and more straightforward to understand.

16. Example 1. 131. Palindrome Partitioning (medium)

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

```

1 For example, given s = "aab",
2   Return
3
4 [
5   ["aa", "b"],
6   ["a", "a", "b"]
7 ]
```

Solution: here we not only need to count all the solutions, we need to record all the solutions. Before using dynamic programming, we can use DFS, and we need a function to see if a splitted substring is palindrome or not. The time complexity for this is $T(n) = T(n-1) + T(n-2) + \dots + T(1) + O(n)$, which gave out the complexity as $O(3^n)$. This is also called backtracking algorithm. The running time is 152 ms.

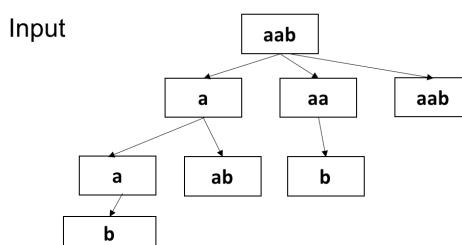


Figure 16.3: State Transfer for the panlindrom splitting

```

1 def partition(self, s):
2     """
3     :type s: str
4     :rtype: List[List[str]]
5     """
6     #s=="bb"
7     #the whole purpose is to find pal, which means it is a
8     #DFS
9     def bPal(s):
10         return s==s[::-1]
11     def helper(s, path, res):
12         if not s:
13             res.append(path)
```

```

13     for i in range(1, len(s)+1):
14         if bPal(s[:i]):
15             helper(s[i:], path+[s[:i]], res)
16     res = []
17     helper(s, [], res)
18     return res

```

Now, we use dynamic programming, for the palindrome, if substring $s(i, j)$ is panlindrome, then if $s[i - 1] == s[j + 1]$, then $s(i-1,j+1)$ is palindrome too. So, for state: $f[i][j]$ denotes if $s[i : j]$ is a palindrome with 1 or 0; for function: $f[i - 1][j + 1] = f[i][j]$, if $s[i] == s[j]$, else ; for initialization: $f[i][i] = \text{True}$ and $f[i][i+1]$, for the loop, we start with size 3, set the start and end index; However, for this problem, this only acts like function $bPal$, checking it in $O(1)$ time. The running time is 146 ms.

```

1 def partition(s):
2     f = [[False for i in range(len(s))] for i in range(len(
s))]]
3
4     for d in range(len(s)):
5         f[d][d] = True
6     for d in range(1, len(s)):
7         f[d-1][d] = (s[d-1]==s[d])
8     for sz in range(3, len(s)+1): #3: 3
9         for i in range(len(s)-sz+1): #the start index , i=0,
0
10            j = i+sz-1 #0+3-1 = 2, 1,1
11            f[i][j] = f[i+1][j-1] if s[i]==s[j] else False
12     res = []
13     def helper(start, path, res):
14         if start==len(s):
15             res.append(path)
16         for i in range(start, len(s)):
17             if f[start][i]:
18                 helper(i+1, path+[s[start:i+1]], res)
19     helper(0, [], res)
20     return res

```

This is actually the example that if we want to print out all the solutions, we need to use DFS and backtracking. It is hard to use dynamic programming and save time.

16.5 Summary

Steps of Solving Dynamic Programming Problems

We read through the problems, most of them are using array or string data structures. We search for key words: "min/max number", "Yes/No" in "subsequence/" type of problems. After this process, we made sure that we

are going to solve this problem with dynamic programming. Then, we use the following steps to solve it:

1. .
2. New storage(a list) f to store the answer, where f_i denotes the answer for the array that starts from 0 and end with i . (Typically, one extra space is needed) This steps implicitly tells us the way we do divide and conquer: we first start with dividing the sequence S into $S_{(1,n)}$ and a_0 . We reason the relation between these elements.
3. We construct a recurrence function using f between subproblems.
4. We initialize the storage and we figure out where in the storage is the final answer ($f[-1]$, $\max(f)$, $\min(f)$, $f[0]$).

Other important points from this chapter.

1. Dynamic programming is an algorithm theory, and divide and conquer + memoization is a way to implement dynamic programming.
2. Dynamic programming starts from initialization state, and deduct the result of current state from previous state till it gets to the final state when we can collect our final answer.
3. The reason that dynamic programming is faster because it avoids repetition computation.
4. Dynamic programming \approx divide and conquer + memoization.

The following table shows the summary of different type of dynamic programming with their four main elements.

	Coordinate Type	Sequence Type	Double Sequence Type	Splitting Type	Backpack Type	Range Type
state	$F[x]$ or $f[x][y]$: state till position (x, y)	$f[i]$: the state till index i , need $n+1$ because we need to consider the empty string	$f[i][j]$: i denotes the previous i number of numbers or characters in the first string, j is the previous j elements for the second string; $[n+1][m+1]$			$f[i][j]$: the state in range $[i,j]$
function	With previous step in the axis with walking	$F[i] = f[j], j=0, \dots, i-1$	$F[i][j]$ to deduct from $f[i-1][j]$, $f[i][j-1]$, $f[i-1][j-1]$			After take one element, if two use min(). eg $f[i][j] = \min(f[i][j-1], f[i-1][j])$
initialize	Normally first column, first row	$F[0]=0, f[1] = \text{nums}[0]$	$f[i][0]$ for the first column and $f[0][j]$ for the first row			When range=1, $i=j$, diagonal
answer	$F[n-1]$ or $\max(f)$ or $f[-1][-1]$	$F[n]$	$f[n][m]$			$F[0][n-1]$
For loop		For i in range(2, $n+1$)				Use for loop to fill in upper diagonal For i in range(size) For start index Get the index j
Space optimization		Rolling vector				

Figure 16.4: Summary of different type of dynamic programming problems

Greedy Algorithms

Greedy algorithm is a further optimization strategy on top of dynamic programming. It usually constructs and tracks a single optimal solution to problem directly and incrementally; like the dynamic programming, it works with subproblems, and at each step it extends the last partial solution by evaluating all available candidates and then pick the best one at the moment without regard to other discarded solutions. Greedy algorithm picks the best immediate output, but does not consider the big picture, hence it is considered greedy.

Because of the “greediness” of the greedy algorithms, whether the single one solution we derive is optimal or not is what for us to ponder and decide. The consciousness of its optimality is important: if we require an absolutely optimal solution, we have to prove its optimality with systematic induction methods, if we are aware that it wont lead to the optimal solution, but is close enough and a good approximation to the optimal solution that we seek but too expensive to achieve, we can still go for it. This chapter is a systematic study of the greedy algorithm, we focus on designing and proving methods that always try to achieve the optimal solution.

Greedy algorithm is highly related to and relies on **math optimization**. It is “easy” if you can reason a solution and prove it easily with math, which comes “natural”. Greedy algorithm can be “hard” when we need to identify important and less obvious properties, design a greedy approach, and prove its correctness with more systematic induction methods; it requires even more analysis effort than dynamic programming does. Because of the highly flexibility of the greedy algorithms, a lot algorithmic books do not even cover this topic. It is not frequently seen in real interviews, but we want to cover it because in the field of AI, the searching is approximate, greedy algorithm

can be approximate too and efficient. Maybe it will inspire us in other fields.

17.1 Exploring

Maximum Non-overlapping Intervals (L435) Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping. Note: You may assume the interval's end point is always bigger than its start point. Intervals like [1,2] and [2,3] have borders “touching” but they don't overlap each other.

Example 1:

Input: [[1,2], [2,3], [3,4], [1,3]]

Output: 1

Explanation: [1,3] can be removed and the rest of intervals are non-overlapping.

Analysis Naively, this is a combination problem that each interval can be taken or not taken, which has a total of $O(2^n)$ combinations. For each combination, we make sure its a feasible that none of the within items overlaps. The process of enumerating the combination has been well explained in the chapter of search and combinatorics. As a routine for optimization problem, we use a sequence X to represent if each item in the original array is chosen or not, $x_i \in \{0, 1\}$. Our objective is to optimize the value:

$$o = \max \sum_{i=0}^{n-1} x_i \quad (17.1)$$

$$(17.2)$$

However, if we sort the items by either start or end time, the checking of an item's compatibility to a combination will be only need to compare it with its last item

Dynamic Programming

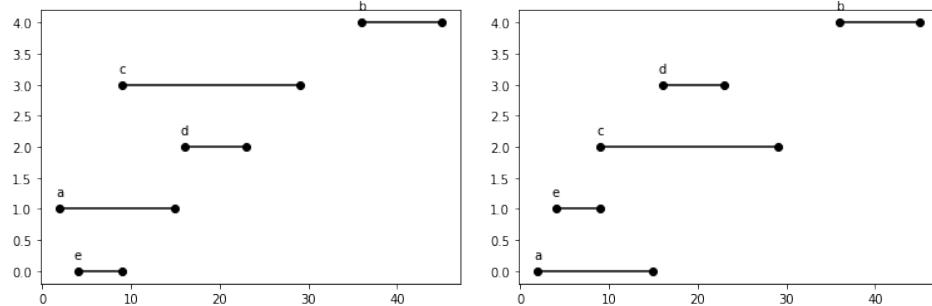


Figure 17.1: All intervals sorted by start and end time.

A-B: Convert to Longest Increasing Subsequence A feasible solution would be that a_0, a_1, \dots, a_k , and $s(a_i) \leq f(a_i) \leq s(a_{i+1}) < f(a_{i+1})$. If we sort the intervals by either start or end time, our sorted intervals as shown in Fig. 17.1. We can reduce our problem into finding the length of longest subsequence $LS, i = [0, k - 1]$ that does not overlap , which is equivalently defining that $s[i + 1] \geq f[i]$ in the resulting subsequence. This is similar enough to the concept of longest increasing subsequence, we can apply the dynamic programming to solve this problem with a time complexity of $O(n^2)$.

For this problem, there can exist multiple optimal solutions and dynamic programming can tell us from the LIS array that which one has the maximum. Let's define a subproblem $d[i]$ as getting the maximum number of non-overlapping intervals for subarray $[a[0], a[1], \dots, a[i - 1]]$ with the maximum subsequence that includes $a[i - 1]$. Then, our the recurrence relation is:

$$d[i] = \max(d[j]) + 1, j \in [0, i - 1], j < i, f(j) < s(i). \quad (17.3)$$

And the answer is $\max(d[i]), i \in [0, n - 1]$.

```

1 from typing import List
2 def eraseOverlapIntervals(intervals: List[List[int]]) -> int:
3     if not intervals:
4         return 0
5     intervals.sort(key=lambda x: x[0])
6     n = len(intervals)
7     LIS = [0] * (n + 1)
8     for i in range(n):
9         max_before = 0
10        for j in range(i, -1, -1):
11            if intervals[i][0] >= intervals[j][1]:
12                max_before = max(max_before, LIS[j + 1])
13        LIS[i + 1] = max(LIS[i], max_before + 1)

```

```

14     #print(LIS)
15     return len(intervals)-max(LIS)

```

Simplified Dynamic Programming Let's approach the problem directly, define a subproblem $d[i]$ as the maximum number of non-overlapping intervals for subarray $a[0 : i]$. With induction, assume we have solved all subproblems from $d[0]$ up till $d[i - 1]$, meaning we have known the answer to all these subproblems. Now we want to find the recurrence relation between subproblem $d[i]$ and its preceding subproblems. We have $a[i]$ at hand, what effect it can have?

We can either increase its previous maximum value which is $d[i - 1]$ by one, or else, the optimal solution remains unchanged. This makes the d array **non-decreasing** sequence. With this characteristic, we do not need to try out all preceding compatible intervals, but instead just its nearest preceding one—because it is at least the same as all preceding ones. We define this preceding compatible interval of $a[i]$ with index $p[i]$, then our recurrence relation become

$$d[i] = \max(d[i - 1], d[p[i]] + 1). \quad (17.4)$$

And the final answer will be $dp[-1]$. With the sorting, the part with the dynamic programming only takes $O(n)$, making the total time $O(n \log n)$ mainly caused by sorting. The Code only differs one line with the above approach:

```

1 def eraseOverlapIntervals(intervals: List[List[int]]) -> int:
2     if not intervals:
3         return 0
4     intervals.sort(key=lambda x: x[0])
5     n = len(intervals)
6     dp = [0] * (n + 1)
7
8     for i in range(n):
9         max_before = 0
10        for j in range(i, -1, -1):
11            if intervals[i][0] >= intervals[j][1]:
12                max_before = max(max_before, dp[j + 1])
13                break
14        dp[i + 1] = max(dp[i], max_before + 1)
15    #print(LIS)
16    return n - dp[-1]

```

Greedy Algorithm

In the previous solution, the process looks like this: If it is sorted by end time, first we have $e, m = 1$, for a , it is not compatible with e , according to previous recurrence relation, $m = 1$, with either a or e in the optimal

solution. When we are processing d , its preceding compatible interval is e , making our maximum value 2 for this subproblem. For c , the length of the optimal solution remains the same, but with additional optimal solution: e, c . However, if we go back, when we are processing a , is it necessary to keep a and e as the optimal solution. If we just get the maximum length of the optimal solution, we do not need to track all optimal solutions, but just one that is the most “optimistic”, which will be e in our case. Because choosing e instead of a leaves more space and thus more likely to fit more intervals for the later subproblems. Similarly, for c , it is incompatible with d , then it is safe to throw it away, because it has the largest end time—the least optimistic, thus it is unnecessary to replace any previous interval in the optimal solution with it. This algorithm takes this simplification even more aggressive and “greedy”. Therefore, in the greedy algorithm, if we have multiple optimal solutions, usually it only cares about **one** that is the most promising and optimistic, and incrementally to build up on it. The code is given:

```

1 def eraseOverlapIntervals(intervals: List[List[int]]) -> int:
2     if not intervals:
3         return 0
4     min_rmv = 0
5     intervals.sort(key = lambda x: x[1])
6     last_end = -sys.maxsize
7     for i in intervals:
8         if i[0] >= last_end: #non-overlap
9             last_end = i[1]
10        else:
11            min_rmv += 1
12
13 return min_rmv

```

If we sort our problems by start time. We need to tweak the code a bit, that whenever one interval is incompatible with previous, we see if it has earlier end time than the previous one, if it is, then we replace it with this one, because it has later start time, and earlier end time, whatever the optimal that the previous interval is in, replacing it with the current one will not overlap and it will be more promising.

```

1 for i in intervals:
2     if i[0] < last_end: #overlap, delete this one, do not update
3         the end
4             if i[1] < last_end:
5                 last_end = i[1]
6                 min_rmv += 1
7             else:
8                 last_end = i[1]

```

Summary and Comparison

We have seen that both dynamic programming and the greedy algorithms solves the problems **incrementally**—starting from small problems to larger problems. Dynamic programming plays safe by tracking the previous state, thus it does not matter how you sort these intervals; both by start time and end time work the same. However, in the greedy approach, it cares less about the previous states.

For example, if our intervals are $[[1, 11], [2, 12], [13, 14], [11, 22]]$, the dynamic programming will give us LIS= $[0, 1, 1, 2, 2]$, which indicates there are two optimal solutions. While, in the greedy algorithm, we would find one that is $[1, 11], [13, 14]$. The resulting is, we might find a solution that is one of its multiple optimal solutions with the same length. In this process, we greatly increased our time efficiency, and simplified the algorithm design and coding.

Questions to Ponder

- Are you absolutely sure that is one of the optimal solutions? If it is optimal, how to prove it then?
- When can I use greedy algorithm over dynamic programming?



What if there each interval is weighted with a real value w_i , and the objective is to maximize a non-overlap set of interval's sum of weights?

If the weight can be both negative and positive, we have to use the first dynamic programming method. If for every $w_i \geq 0$, then previous suboptimal solution can still have a chance to lead to a global optimal solution if it happens to be compatible with following intervals with large weight, we can apply the second dynamic programming method.

17.2 Introduction to Greedy Algorithm

What is Greedy Algorithm?

We say that dynamic programming tracks best solution to all subproblems (in the above example, it is the `d` array) and incrementally build up solutions to subproblems using their subproblems (in our example, we use all of its subproblems `for j in range(i)` to build up solution to subproblem `d[i]`).

Greedy algorithm follows the same trend in the sense of solving overlapping subproblems where optimal substructure property shows. But it only maintain **one optimal solution** for each of the subproblem—the most

promising one. For example, for $[[1, 11]]$, the optimal solution is $[1, 11]$, for $[1, 11], [2, 12]$, the optimal solution is still $[1, 11]$, even though $[2, 12]$ is another optimal solution for this subproblem. For $[1, 11], [2, 12], [13, 14], [11, 22]$, greedy approach gives us $[1, 11], [13, 14]$ as our optimal solution, while in dynamic programming, we can still find another optimal solution: $[1, 11], [11, 22]$.

Three Properties We define three properties for greedy algorithm:

- Overlapping Subproblems and Optimal substructure property: These two properties defined exactly the same as in dynamic programming. If an optimal solution to the problem contains within it optimal solutions to its subproblem, this is said to be optimal substructure. In our example, $[1, 11], [2, 12], [13, 14]$, the optimal solution $[1, 11], [13, 14]$ contains optimal solution $[1, 11]$ that is to its subproblem $[1, 11], [2, 12]$.
- Greedy-choice property: This is the only additional property that greedy algorithm holds compared with dynamic programming. We can assemble a globally optimal solution by making a locally optimal (greedy) choice.

For example, given an array $[2, 1, 3, 7, 5, 6]$, which has as $[1, 3, 5, 6]$, $[2, 3, 5, 6]$ as the longest increasing subsequence. We define the LIS as the longest increasing subsequence that ends at $a[i-1]$ for array $a[0 : i]$. The process of constructing it with dynamic programming shows as follows:

```

1      subproblems
2      [2] , LIS = [2]
3      [2, 1] , LIS = [1]
4      [2, 1, 3] , LIS= [1, 3], [2, 3]
5      [2, 1, 3, 7] , LIS = [1, 3, 7], [2, 3, 7]
6      [2, 1, 3, 7, 5] , LIS = [1, 3, 5], [2, 3, 5]
7      [2, 1, 3, 7, 5, 6] , LIS = [1, 3, 5, 6], [2, 3, 5, 6]
8

```

We clearly see that to get the best solution, we have to rely on the optimal solution of all preceding subproblems. If we insist on applying greedy algorithm, this is how the process looks like:

```

1      subproblems
2      [2] , LIS = [2]
3      [2, 1] , LIS = [2] , only compare [2] and 1
4      [2, 1, 3] , LIS= [2, 3]
5      [2, 1, 3, 7] , LIS = [2, 3, 7]
6      [2, 1, 3, 7, 5] , LIS = [2, 3, 7]
7      [2, 1, 3, 7, 5, 6] , LIS = [2, 3, 7]
8

```

LIS = [2, 3, 7] is locally optimal but not part of the global optimal solutions which are [1, 3, 5, 6] and [2, 3, 5, 6]. In our non-overlapping interval problem, if one interval is optimal in the local subproblem, it will sure be part of the optimal solution to the final problem (globally).

To summarize, greedy algorithms simply works on incrementally build up one optimal solution. For this single optimal solution to be globally optimal, each partial optimal solution has to exactly match some prefix of the optimal solution. Both proving the correctness and design of greedy algorithm thus has to be done by induction and that at each stage, greedy algorithm is making the best choices.

To correctly design a greedy algorithm, it has to make a locally optimal choice according to some rules or orderings. In the above example, we know the optional solution has the property that $s_i \leq f_i, f_i \leq f_{i+1}$. By sorting the intervals with increasing order of the finish time. The greedy approach choose the interval with the earliest finish time, and it says, this belongs to my optimal solution. And it just need to go through all the candidates in the order of finishing time and see if it is compatible with the last item, and we would build up a feasible and optimal solution. It orders its subproblems to make sure each partial optimal solution will be “prefix” or part of the global optimal solution.

Practical Guideline

It is clear to us like in dynamic programming, greedy algorithms are for solving optimization problems, and it subjects to a set of constraints. For example:

- Maximize the number of events you can attend, but do not attend any overlapping events.
- Minimize the number of jumps
- Minimize the cost of all edges chosen, but do not disconnect the graph.

Do not worry about the definition of greedy algorithm; it is hard and often confusing, because it is a natural and highly dependable on the problem context. However, to come up with a rule that.

I suggest we start with dynamic programming, it is more systemized, easier to prove the correctness, and it guides us to walk through to the greedy algorithm which is more efficient just as the process shown in the example.

Ordering, Monotone Property These constraints bring sense of ordering in our optimal solution, and this is when greedy algorithm applies. Therefore, we say:“beneath every greedy algorithm, there is almost always

a more cumbersome dynamic programming solutions". But, not every dynamic programming we can find a more efficient greedy algorithm, because to make greedy algorithm work, there needs to have some ordering in the optimal solution that makes the locally optimization applicable and globally optimal. We shall see this in our examples! In the activity scheduling, it is that $d[i]$ is non-decreasing as shown in its dynamic programming solution. Because of this property, a global therefore, and in the Dijkstra's algorithm, it is that $w(s, u) < w(s, v) = w(s, u) + w(u, v)$. in the shortest path. Once this monotone property breaks as in a graph with negative weights, greedy algorithm won't apply and we have to retreat to dynamic programming.

Pros and Cons

As we see, greedy algorithm has the following pros:

- Simplicity: Greedy algorithms are often easier to describe and code up than other algorithms.
- Efficiency: Greedy algorithms can often be implemented more efficiently than other algorithms.

However,

- Hard to get it right: Once you have found the right greedy approach, designing greedy algorithms can be easy. However, finding the right rule can be hard.
- Hard to verify/prove: Showing a greedy algorithm is correct often requires a nuanced argument.

17.3 *Proof

The main challenging in greedy algorithms is to prove its correctness, which is important in theoretical study. However, in real coding practice, we can leverage the dynamic programming solution to compare with and scrutinize different kinds of examples to make sure the greedy algorithm and the dynamic programming are having the same results. Still, let us just learn this proof techniques as mastering another powerful tool.

17.3.1 Introduction

First, we introduce generally two techniques/arguments to prove the correctness of a greedy algorithm in a step-by-step fashion using the mathematical induction, they are: **Greedy Stays Ahead** and **Exchange Arguments**.

Greedy stays ahead

This simple style of proof works by showing that, according to some measures, the optimal solution built by the greedy algorithm is always at least or better than the optimal solution during each iteration of the algorithm. Once we have established this argument, we can show that the greedy solution must be optimal. Typically there are four steps:

1. Define the solution: Define our greedy solution as G and we compare it against some optimal solution O^* .
2. Define the measurement: Your goal is to find a series of measurements you can make of your solution and the optimal solution. Define some series of measures $m_1(X), m_2(X), \dots, m_n(X)$ such that $m_1(X^*), m_2(X^*), \dots, m_k(X^*)$ is also defined for some choices of m and n . Note that there might be a different number of measures for X and X^* , since you can't assume at this point that X is optimal.
3. Prove Greedy Stays Ahead: Prove that $m_i(X) \geq m_i(X^*)$ or that $m_i(X) \leq m_i(X^*)$, whichever is appropriate, for all reasonable values of i . This argument is usually done inductively.
4. Prove Optimality. Using the fact that greedy stays ahead, prove that the greedy algorithm must produce an optimal solution. This argument is often done by contradiction by assuming the greedy solution isn't optimal and using the fact that greedy stays ahead to derive a contradiction.

The main challenge with this style of argument is finding the right measurements to make.

Exchange Arguments

It proves that the greedy solution is optimal by showing that we can iteratively **transform** any optimal solution into the greedy solution produced by greedy algorithm without worsening the cost of the optimal solution. This transformability matches the word “exchange”. Exchange arguments are a more versatile technique compared with greedy stays ahead. It can be generalized into three steps:

1. Define the solution: Define our greedy solution as $G = \{g_1, \dots, g_k\}$ and we compare it against some optimal solution $O = \{o_1, \dots, o_m\}$.
2. Compare solutions: Assume the optimal solution is not the same as the greedy solution; show that if $m(G) \neq m(O)$, then G and O must differ in some way. How it differs depend on the measurement and the problem context.

- (a) If it is a combination and a length problem, then $m(G) = k$, and $m(O) = m$, we need to prove $k = m$.
 - (b) If it is a combination and with objective as a value, we assume o_1 and g_1 differs and all others are identical. Then, we swap o_1 with g_1 .
 - (c) If it is a permutation with objective function, we assume there are two consecutive items in O that is in a different order than they are in G (i.e. there is inversion)
3. Exchange Arguments: Show how to transform O by exchanging some piece of O for some piece of G . Then, we prove that by doing so, we did not increase/decrease the cost of O as we transform O to G with more iterations, proving that greedy is just as good as any optimal solution and hence is optimal.

Guideline

We will simply go through the list and, but the point is it is we should use the proof methods as a way to design the greedy algorithm on top of the dynamic programming.

17.3.2 Greedy Stays Ahead

Maximum Non-overlapping Intervals Proofs Let's use G, O for our greedy and optimal solution respectively. G consists of $\{G_1, G_2, \dots, G_k\}$ in the order they each item is added into the G . Similarly, O_1, O_2, \dots, O_m is one of the optimal sets.

17.3.1 G is a compatible set of intervals.

G is trivially feasible because in our design we discard any interval that overlaps with our previous greedy choice. Now, all it matters is to prove its optimality.

We know that there might exist multiple optimal solutions for a problem, just as shown in our example. In this particular problem, we do not intend to prove that $G = O$, because it might not; we prove that G and O has the same length instead $|G| = |O|$. We will apply “greedy stays ahead” principle along with mathematical induction. We first assume that the intervals in O is ordered by the same rule applied on $G - f(O_i) < f(O_j)$, if $i < j$. In this case, we use the finishing time f of each interval to measure. To prove $|G| = |O|$, we need to prove Greedy stays ahead and the optimality.

17.3.2 $f(G_i) \leq f(O_i), i \in [1, k]$,

For $i = 1$, the statement is true: our algorithm starts by choosing interval with minimum finish time. We further assume the statement is true for

$n - 1$ as our induction hypothesis. 17.3.2 will be proved if we can prove that $f(G_n) \leq f(O_n)$.

$f(G_{n-1}) \leq f(O_{n-1}) \leq s(O_n)$ tells us: right after the selection of G_{n-1} , O_n together with others remains in the available set for the selection at step n . $f(O_{n-1}) \leq s(O_n)$ is always true because O is a compatible set, and naturally, $f(O_n) > s(O_n) \geq f(O_{n-1})$. The greedy algorithm selects the available interval with smallest finish time, which guarantee that $f(G_n) \leq f(O_n)$. Now, we have formally proved our sense of “greedy stays ahead” or rather “greedy never fall behind” using induction.

17.3.3 G is optimal: $|G| = |O|$.

We apply contradiction: if G is not optimal, then we must have $m > k$. Similarly, after step k , we have $f(G_k) \leq f(O_k) \leq s(O_{k+1})$, and O_{k+1} must be remaining in the available set for greedy algorithm to choose. Because greedy algorithm only stops when the remaining set is empty—a contradiction. So far, we successfully applied the “greedy stays ahead” method to prove the correctness in the example of maximum non-overlapping intervals.

17.3.3 Exchange Arguments

Scheduling to minimize lateness: Instead of having a fixed start and end time for each interval, we relax the start time. Therefore, we represent the interval with $[t_i, d_i]$, where t_i is the contiguous time interval and d_i is the deadline. There are many objective functions we might want to optimize. Here, we assume we only have one resource, what is the maximum number of meetings that we can schedule into a single conference room and none of them is late or we allow some meetings to run late, but define lateness l_i as:

$$l[i] = f[i] - d[i] \quad (17.5)$$

Say, our object is to minimize the total lateness scheduling all meetings in one conference room, find the optimal solution:

$$O = \min \sum_{i=0}^{n-1} l_i \quad (17.6)$$

$$= \min \sum_{i=0}^{n-1} f[i] - d[i] \quad (17.7)$$

```
For example, we have nums = [[4, 6], [2, 6], [5, 5]]
The optimal solution is [2, 6], [4, 6], [5, 5] with total
lateness (2-6)+(2+4-6)+(2+4+5)-5 = -4+0+6 = 2
```

Example 2:

```
nums = [[2, 15], [36, 45], [9, 29], [16, 23], [7, 9], [4, 9], [5, 9]]
ans = 47
```

Analysis First, let us assume all intervals have distinct deadline. A naive solution is to try all permutation of n intervals and find the one with the minimum lateness. But what if we start from random order, we compute its lateness and each time we exchange two adjacent items and see if this change will decrease the total lateness or not.

```
1 a i a j
```

Therefore, Say our items are a_i and a_j . There are four cases according to d_i, d_j, t_i, t_j . At first, with lateness $s + t_i - d_i$, and $s + t_i + t_j - d_j$. After the exchange, we have $s + t_j - d_j$ and $s + t_j + t_i - d_i$. i will definitely be more late, j however will be less late. Let us compare the additional lateness of i with the decreased lateness of j :

$$s + t_j + t_i - d_i - (s + t_i - d_i) \rightarrow s + t_i + t_j - d_j - (s + t_j - d_j) \quad (17.8)$$

$$t_j \rightarrow t_i \quad (17.9)$$

Therefore, we have to exchange i, j if $t_j < t_i$. Thus, ordering the list with increasing order of duration time of each meeting will have the best objective. Our Python code is:

```
1 def lateness(intervals):
2     intervals = sorted(intervals, key=lambda x: (x[0]))
3     f = 0
4     ans = 0
5     for i, (t, d) in enumerate(intervals):
6         f += t
7         ans = ans + f-d
8     return ans
```

Modification However, if we modify our definition of lateness as:

$$l[i] = \begin{cases} 0, & \text{if } f[i] \leq d[i], \\ f[i] - d[i], & \text{otherwise.} \end{cases} \quad (17.10)$$

Which is to say we do not reward for intervals that are not late with negative values. Things get more complex.

1. If none of them is late, then exchange or not to change will not make any difference to the total lateness.

```
1 a i a j d i d j
```

2. If both is late, then exchange the items if $t_j < t_i$.
3. If i is late, and j is not late, then no matter about their t , exchange them will only be even later.
4. If i is not late, and j is late. Exchange them will totally depends

Therefore, if we change the definition of lateness, greedy solution is no longer available for us, not even dynamic programming. But the greedy approach that first sorts the intervals by the duration time will get us a good start, then we can track the smallest lateness with backtracking and search prune by its minimum lateness found so far.

```

1 def lateness(intervals, f, l, globalmin, globalans, ans, used):
2     if len(ans) == len(intervals):
3         if l < globalmin[0]:
4             globalmin[0] = l
5             globalans[0] = ans[::-1]
6             return
7     for i, (t, d) in enumerate(intervals):
8         if used[i]:
9             continue
10        used[i] = True
11        f += t
12        if f-d >= 0:
13            l+= (f-d)
14        if l < globalmin[0]:
15            ans.append(i)
16            lateness(intervals, f, l, globalmin, globalans, ans, used)
17            ans.pop()
18        if f-d >=0:
19            l -= (f-d)
20        f -= t
21        used[i] = False
22    return

```

We call this function with code:

```

1 intervals = sorted(intervals, key=lambda x: (x[0]))
2 globalmin, globalans = [float('inf')], []
3 ans = []
4 used = [False]*len(intervals)
5 lateness(intervals, 0, 0, globalmin, globalans, ans, used)
6 print(globalmin, globalans[0])
7 for i in globalans[0]:
8     print(intervals[i], end=' ')

```

We will get the following output:

```

63 [1, 2, 0, 3, 4, 5, 6]
[4, 9] [5, 9] [2, 15] [7, 9] [9, 29] [16, 23] [36, 45]

```

We can see that no particular rule—not sorting by d , not by t , and not by $d-t$ which is called **slack time**—we can find that to solve it in greedy polynomial time. However, can we use dynamic programming?

Dynamic Programming We can first do a simple experiment: we take out $[2, 15]$ from our `intervals`, the resulting optimal solution keeps the same order as $[4, 9] [5, 9] [7, 9] [9, 29] [16, 23] [36, 45]$, which is a very good indicator that dynamic programming might apply. We can keep taking out

and the optimal solution is still simply the same order, this indicates the optimal substructure.

Let us assume we find the best order O for subarray $\text{intervals}[0 : i]$, now we have to prove that the best solution for subarray $\text{intervals}[0 : i + 1]$ can be obtained by inserting `interval[i]` into O . Assume the position we insert is at j , so $O[0 : j]$ will not be affected at all, we care about $O[j : i]$. First, we have to prove that no matter where to add insert `interval[i]`, the ordering of O needs to keep unchanged for it to have optimal solution. If insert position is at the end of O , the ordering do not need to change. For the other positions, however, it is really difficult to prove without enough math knowledge and optimization.

Let us assume the start time is s for $j, j + 1$, we know:

$$l(s + t_j - d_j) + l(s + t_j + t_{j+1} - d_{j+1}) \leq l(s + t_{j+1} - d_{j+1}) + l(s + t_j + t_{j+1} - d_j) \quad (17.11)$$

Because $l(c) \in [0, c]$, prove that

$$l(s + t_j + t_i - d_j) + l(s + t_j + t_i + t_{j+1} - d_{j+1}) \leq l(s + t_{j+1} + t_i - d_{j+1}) + l(s + t_j + t_i + t_{j+1} - d_j) \quad (17.12)$$

We can not prove it, and we use this method to try out, but it gives us wrong answer, so far, all our attempt to use greedy algorithm failed miserably.

When there is a tie at the deadline, if we schedule the one that takes the most time first, we end up with higher lateness. For example, if our solution is $[5, 5], [4, 6], [2, 6]$, the lateness is $5+4-6 + (9+2-6) = 3+5 = 8$ instead of 6.



What if each interval, if we are allowing multiple resources, what is the least number of conference rooms we need to schedule all meeting.



630. Course Schedule III, find the maximum number of non-overlapping meetings can be scheduled within one resource.

Prove the Kruskal's Algorithm Let the optimal minimum spanning tree be $O = (V, E^*)$, and the one generated by greedy approach be $G = (V, E)$. $|E^*| = |E|$ because both is a tree and the number of edges always equal to $|V| - 1$. Assume there is one edge $e \in E^*, e \notin E$, this means that there is another edge $f \in E$ that differs from e . Other than these two edges,

all the other edges are the same. For example, in the graph we say $e = (1, 5)$. With the constraint that there is only edge differs, f has to be one edge out of $(2, 3), (3, 5)$; adding e to T forms a cycle, so in T^* , it can not have edges $(2, 3), (3, 5)$ at the same time, thus one referred as f has to be removed in the T^* . It is always true that $\text{cost}(e) \geq \text{cost}(f)$, because otherwise the greedy approach would have chosen e instead of f .

For the optimal approach, if we replace e with f , then we have $\text{cost}(T) = \text{cost}(T^* - e + f) \leq \text{cost}(T^*)$. This means, with this swap of e and f between G and O , the cost of the greedy approach is still at most the same as the optimal cost, transforming the optimal solution to greedy solution will not worsen the optimal solution.

17.4 Design Greedy Algorithm

We have seen greedy algorithm design, definition, and different examples of greedy approaches and its proof. One obvious sign that states greedy approach might apply is, “sorting will not incur the correctness of the optimal solution but rather greatly simplify the design complexity”. Generally, people design a greedy algorithm by trying out rules with objection and hopefully find one good enough and then prove its correctness. This approach is simple but fuzzy. Thus, we prefer a more systemized design approach:

1. Search: Analyze the problem with search and combination—no implementation is needed, to know our atomic search complexity.
2. Dynamic Programming: Design dynamic programming approach first by defining state and constructing a recurrence relation repeatedly until we find one that works well. This step brings us closer to the greedy approach: it gives us definition of state, recurrence relation and a polynomial time complexity.
3. Greedy: then further to see if the greedy choice property holds—between a subproblem p_i and its succeeding subproblem p_{i+1} , if the optimal solution within p_i is also part of the optimal solution within p_{i+1} or we can simply construct an optimal solution from previous optimal solutions without checking multiple subproblems. If it holds, great, the previous dynamic programming becomes an overkill, and we further improve our approach by simplifying the recurrence relation with “rules”, which saves us time and/or space. To derive a good “rule”, we have to study and understand a bunch of “facts”, thus strengthen our choice with: Does the greedy optimal solution always stay ahead and be the most promising optimal solution at each step? If not, try exchanging some items within the previous optimal solution and see if it improves the situation and keeps us staying ahead.

We will solve the following classical problems with this

17.5 Classical Problems

List classical problems

17.5.1 Scheduling

We have seen two scheduling problems, it is a time-based problem which naturally follows has a leftmost to rightmost order along the timeline. And it is about scheduling tasks/meetings to allowed resources. We need to pay attention to the following contexts:

- Do we have to assign all intervals or just select a maximum set from all? This relates to the number of resources that are available.
- What are the conditions? Is both start and end time fixed, or they are highly flexible and are bounded by earliest possible start time and latest end time?

The core principle is to answer these questions:

- Start and end time:
 - Is both fixed? Yes, then we are simply giving these intervals as a state, no change at all, and the ordering of the start and end time is exactly the same. If the question is only given one resource and we need to get the maximum non-overlapping intervals, easy piece of cake, we follows the order, and check if it is compatible with one previous intervals. If it asks about the minimum resources needed, that is the depth of the set, that is the property, we have discussed how assigning a meeting room to a preceding free meeting room does not affect the number of free rooms for the next.
 - If it is not fixed, we are given either t_i and d_i for each interval—we can start at any time s_i , finish at $s_i + t_i$, but we do have a deadline d_i that better to be met—or we are given b_i, t_i, d_i —we can start at any s_i , but it would better be $s_i \geq b_i$, and end at $s_i + t_i \leq d_i$. (The second is not sure). The fundamental rule is: **Earliest Deadline First**. We have proved that if there is an inversion, swapping them will only result better objective value. This usually points out that the optimal solution shall have no inversion and no idle time on the resource. Whenever we met a tie at the deadlines, no matter what order of these intervals with equal deadline, the total lateness is usually the same, which can be proved with inversion.

Scheduling all Intervals(L253. Meeting Rooms II) In our previous scheduling problem, there is only a single resource to fit in non-overlapping intervals. Scheduling all intervals on the other hand requires us to schedule all the intervals with as few resources as possible. This problem is also known as **interval partitioning problem** or **interval coloring problem** because our goal is to partition all intervals across multiple resources, and it is like each resource to be assigned a color.

Given an array of meeting time intervals consisting of start and end times $[[s_1, e_1], [s_2, e_2], \dots] (s_i < e_i)$, find the minimum number of conference rooms required and assign a label for each interval.

Example 1:

```
Input: [[2,15, 'a'], [36,45, 'b'], [9,29, 'c'], [16,23, 'd'], [4,9, 'e']]
Output: 2
```

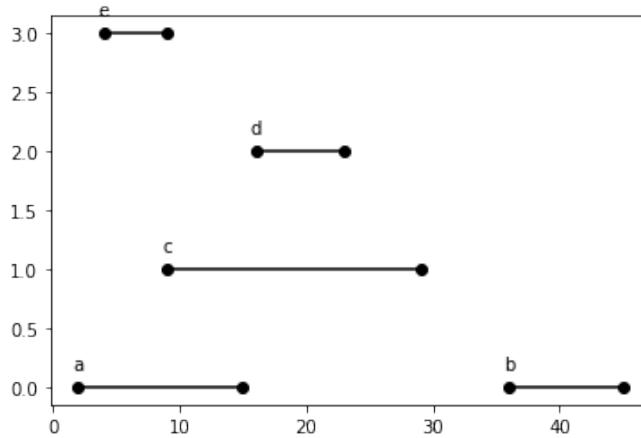


Figure 17.2: All intervals

Analysis The example is plotted in Fig. 17.2. A universal solution is to treat each interval as a vertex, if two intervals overlap, connect them with an edge, thus forming a graph. Now, the problem is reduced to a graph coloring, however, it might be to complicating things.

Find Minimum Number of Conference Rooms

First, let us solve the first problem: What is the minimum number of conference rooms required? By observation and intuition, if at the same time point, there are a number of overlapping intervals, we have to assign each of these intervals a different resources. Now, we define the depth d as the maximum number of overlapping intervals at any single point on the time-line, we claim:

17.5.1 In the interval partitioning problem, the number of resources needed is at least the depth d of the set of intervals.

Before we head off to the proof, let's discuss how to find the depth. According to the definition, if we are lucky that the time given is in the form of integer, then, the most straightforward way is to use the **sweep line** method, with a **counter** to track the number of intervals at each integer time moment. We use a vertical line to sweep from the leftmost to the right most intervals: this exactly follows a natural order, that when the earliest meeting starts, we have to assign a room to it no matter what (start has $+1$), and when can reuse a meeting room assigned before only if there is one that is freed (end has -1 as value). The sweep line method We need to watch out for the edge case, when two intervals where the finish time of one and the start time of the other overlaps, such as $[4, 9], [9, 29]$, this is not counted as two. Therefore we make sure to exclude the finish time when scanning, in range $[s, e)$.

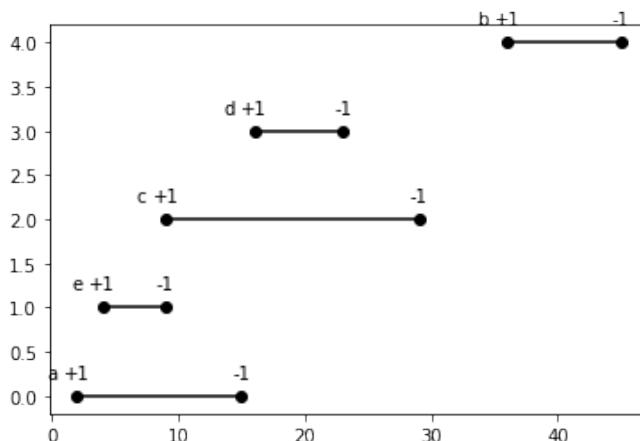


Figure 17.3: All intervals sorted by start and end time.

However, this process can be simplified. When we are scanning, we will notice that the count will only change when encountering start or finish time point. First, at the start time of a , we have to assign one room, then at the start time of e , we have to assign a second room, and at the end of e , we free the room, and at the start time of c , it reuses the second room right away. Then in the line, at the end of a , it releases room 1, so d starts reusing the first room right away. We can simply assign 1 to the start time, -1 to the finish time, and put all of these points into a list, sort them by time first. Because to handle the edge case—a tie in the previous sort where the start and end time is the same, the second degree sorting is used to put -1 in front of 1 to avoid overcounting the rooms. This process is shown in Fig. 17.3.

```
1 def minMeetingRooms( intervals ) :
```

```

2     if not intervals:
3         return 0
4     points = []
5     for s, e in intervals:
6         points.append((s, +1))
7         points.append((e, -1))
8     points = sorted(points, key=lambda x: (x[0], x[1]))
9     ans = 0
10    total = 0
11    for _, flag in points:
12        total += flag
13        ans = max(ans, total)
14    return ans

```

Label Assignment

We can modify the previous code to incorporate label assignment. We separate the start and end time in two independent lists because only when we meet a start time, we assign a room, and sort both of them.

```

2(0) 4(4) 9(2) 16(3) 36(1)
9(4) 15(0) 23(3) 29(2) 45(1)

```

We put two pointers, sp, ep at the start of the start time list and end time list respectively. We need zero room at first. And for start pointer at 2, we assign room one to interval 0 because $2 < 9$, no room is freed to reuse. Then sp moves to 4. $4 < 9$, no room is freed, assign room 2 to interval 4. sp at 9, $9 \geq 9$, meaning we can reuse the room belonged to interval 4, thus assign room 2 to interval 2. Now, move both sp and ep , we are comparing $16 > 15$, meaning interval 3 can reuse the room belonged to interval 0, we assign room 1 to interval 3. Next, we compare $36 > 23$, interval 1 takes the room number 1 from interval 3. Since one of the pointer reached to the end of the list, process ends.

```

1 def minMeetingRooms(intervals):
2     starts, ends = [], []
3     for i, (s, e) in enumerate(intervals):
4         starts.append((s, i))
5         ends.append((e, i))
6     starts.sort(key=lambda x: x[0])
7     ends.sort(key=lambda x: x[0])
8     n = len(intervals)
9     rooms = [0] * n
10    sp, ep = 0, 0
11    label = 0
12    while sp < n:
13        index = starts[sp][1]
14        # Assign a new room
15        if starts[sp][0] < ends[ep][0]:
16            rooms[index] = label
17            label += 1

```

```

18     else: #Reuse a room
19         room_of_end = rooms[ends[ep][1]]
20         rooms[index] = room_of_end
21         ep += 1
22         sp += 1
23     print(rooms)
24     return label

```

The above method is natural but indeed greedy! We sort the intervals by start time, the worst case we assign each meeting a different room. However, the number of room can be reduced if we can reuse any previous assigned meeting rooms that is free at the moment. **The depth is controlled by the time line.** For example, for interval c , if both a and e is free at that moment, does it matter which meeting room to put of c in? Nope. Because no matter which room it is in, the interval d will overlap with this interval, thus can not use its meeting room, but still there is the one left from either a or e . This is what this problem is essentially different from the maximum non-overlapping problems. The greedy part is we always reassign the room belongs to the earliest available rooms. A non-greedy and naive way is to check all preceding meeting rooms, and find one available.

You have to property Did you find that, for the resource assignment, mostly, we have no much choice, because we have to assign it. The only choice is which room. We are greedy that we merge it whenever we can. All the solutions no matter if they put the earliest finished meeting room to reassign or just random or arbitrary one, they are doing it for a single purpose: reduce the possible number of resources whenever they can.

An easy way to understand this problem is to notice that: for each meeting you HAVE TO assign it a room. The worst case is we assign a room for each single meeting and we do not even need to sort these intervals. Well, how can we optimize it, minimize the number of rooms? We have to reuse a room whenever it is possible. Therefore, we need to sort the meeting by start time. Because the first meeting has no choice but to assign a meeting room to it. For the second meeting, we have two options: either assign a room or reuse one that is available now.

- If I choose to reuse a room, does it influence my optimal solution later on? No, because if we chose to reuse a room, we decrease the total number of room by one, and later on it wont even affect the available rooms for the next meeting. It's like, here is a candy, take it and it wont affect your chance of having candy at all! Of course I would go for it.
- Does it matter which one to reuse? Nope. Why? Because the smallest number of rooms needed are decided by how many meetings collide at a single time point. No matter which available room you put of this

meeting, for the following meetings the number of available rooms are always the same: any rooms that are freed from preceding meetings. Here is the thing. When we are scanning from the leftmost interval to the rightmost by start time,

This is why there are so many different approaches: iterating preceding meetings and find any one that is available or put it into a min-heap to use the earliest available rooms or as the second solution, it is still the same as of the min-heap, reassign one that ends earliest. This optimization process is natural and GREEDY!

Proof We have been proved it already informally. The greedy we have will end up with compatible/feasible solutions where at each meeting rooms, no two overlapping meetings will be scheduled.

17.5.2 *If we use the greedy algorithm above, we can schedule every interval with d number of resources, which is optimal.*

We know that using d number of resources, we have to prove that we can schedule these intervals with d resources.

Organize Second, how to assign a label to each meeting? Actually, it is not necessary to know the number of the minimum conference rooms d needed to assign a label to each, we can get the d by counting the total number of labels. Now, back to be greedy, it might be tempting at first to follow the non-overlapping scheduling problems. First, we sort the intervals by the finish time, and an intuitive strategy to assign labels is: go through intervals in order, assign each interval a label that differs from any previous overlapping interval's. The code is:

```

1 def colorInterval(intervals):
2     intervals = sorted(intervals, key=lambda x: x[1])
3     labels = [] # label list to sorted intervals
4     n = len(intervals)
5     for i, (s, e) in enumerate(intervals):
6         excluded = []
7         for j in range(i):
8             if s < intervals[j][1]: # overlap
9                 excluded.append(labels[j])
10    # assign label
11    for l in range(n):
12        if l not in excluded:
13            labels.append(l)
14            break
15 return len(set(labels))

```

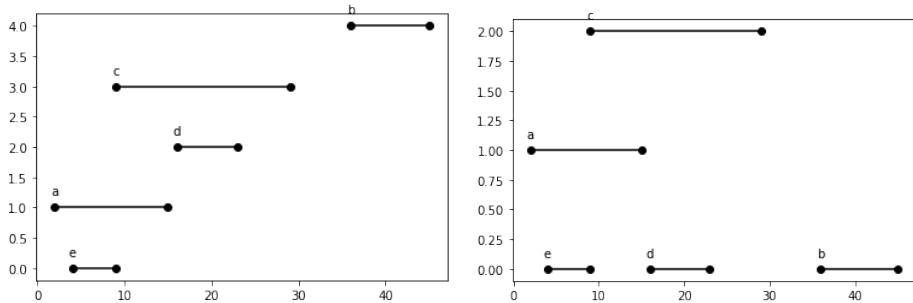


Figure 17.4: Left: sort by start time, Right: sort by finish time.

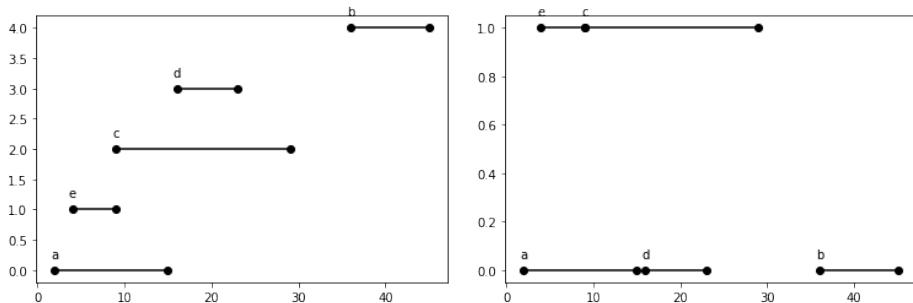


Figure 17.5: Left: sort by start time, Right: sort by finish time.

Unfortunately, it gives us wrong answer; it used three resources instead of 2 as we proved before. The sorting and the answer is plotted in Fig. 17.4. What went wrong? In the non-overlapping interval scheduling problem, what matters is the maximum number of intervals we can fit in within one resource, sorting by finish time guarantees we fit as many intervals as possible. However, in this problem, we try to use as less as possible of resources, we want each resource to be as tight as possible. In math, Therefore, the right way of sorting is to sort by the starting time. If you insist on sorting by finish time, you will get right answer if you traverse the intervals in reversed order.

This type of assignment takes $O(n^2)$ in time complexity. We can easily do better but with the same greedy strategy.

Optimizations

We use a list `rooms` which starts as being empty. Each room, we only keep its end time. After sorting by the start time, we go through each interval and try to put it in a room if it does not overlap, we put this interval in this room and update its end time. If no available room found, we assign a new room instead. With this strategy, we end up with $O(nd)$ in time complexity. When d is small enough, it saves more time.

```

1 def minMeetingRooms(intervals):
2     intervals = sorted(intervals, key=lambda x: x[0])
3     rooms = [] # a list that tracks the end time
4     for s, e in intervals:
5         bFound = False
6         for i, re in enumerate(rooms):
7             if s >= re:
8                 rooms[i] = e
9                 bFound = True
10                break
11            if not bFound:
12                rooms.append(e)
13    return len(rooms)

```

Priority Queue Is there a way to fully get rid of the factor of d ? In our case, we loop over all rooms and check if it is available, but we do not even care which one is, we just need one! So, instead we replace `rooms` with a priority queue with uses min-heap, making sure each time we only check the room with the earliest end time; if it does not overlap, put this meeting into this room and update its finish time, or else assign a new room.

```

1 import heapq
2 def minMeetingRooms(intervals):
3     intervals = sorted(intervals, key=lambda x: x[0])
4     rooms = [] # a list that tracks the end time of each room
5
6     for s, e in intervals:
7         bFound = False
8         # now, just check the room that ends earlier instead of
9         # check it all
10        if rooms and rooms[0] <= s:
11            heapq.heappop(rooms)
12        heapq.heappush(rooms, e)
13    return len(rooms)

```

17.5.2 Partition

763. Partition Labels A string S of lowercase letters is given. We want to partition this string into as many parts as possible so that each letter appears in at most one part, and return a list of integers representing the size of these parts.

Example 1:

Input: $S = "ababc bacade fegde hijh klij"$

Output: $[9, 7, 8]$

Explanation:

The partition is "ababc bacade", "fegde", "hijh klij".

This is a partition so that each letter appears in at most one part.

A partition like "ababcbacadebegde", "hijhklj" is incorrect, because it splits S into less parts.

for example ``abaefegdehi''.
 $\{aba\}, \{efegde\}, \{hi\}$

Analysis We know we can use the partition type of dynamic programming to find the maximum length with

$$d[n] = p[i : n], d[i] \quad (17.13)$$

$$d[n] = \max(d[i] + 1), i < n, \text{and } p[i:n] \text{ is an independent part.} \quad (17.14)$$

$$d[i] = p[j : i], d[j], i \in [0, n - 1], j < i, \quad (17.15)$$

$$d[i] = \max(d[j] + 1) \quad (17.16)$$

This will give us a solution with $O(n^2)$. Not bad! In dynamic programming solution, when we are solving subproblem "abaefegdeh", we are checking all previous subproblems' solutions. However, if we observe, we only need the optimal solutions of the previous subproblem "abaefegde"— "aba", "efegde" to figure out the optimal solution of current: simply check if 'h' is in any parts of the preceding optimal solution and merging parts between the earliest part that incorporates 'h' to the last. Now, we can also observe that between subproblems for example "abaefegdehi".

```
a, o = {a}
ab, o = {a}, {b}
aba, merge, o = {a, b}
abae, o = {a, b}, {e}
abaef, o = {a, b}, {e}, {f}
abaefe, e exists, merge, o = {a, b}, {e, f}
abaefeg
abaefegd
abaefegde
abaefegdeh
abaefegdehi
```

This indeed is already a greedy algorithm. We no longer just blindly care about the subproblem state, but we explicitly working on a single partial optimal solution. We incrementally construct the partial optimal solution till it is optimal globally. Moreover, we can detect it has unnecessary work to track these optimal solutions, we can utilize a `dict` structure to track the last location of a character is seen in the string. We loop through the characters in the string one by one, and track a subarray's start and end location in the string. Whenever a new character comes, it can either enlarge

this part, or within the last range, or it is the end of the range and different type of process is applied for different case. The Python code shows more details of this greedy approach algorithm:

```

1 from collections import defaultdict
2 class Solution:
3     def partitionLabels(self, S: str) -> List[int]:
4         n = len(S)
5         loc = defaultdict(int)
6         for i, c in enumerate(S):
7             loc[c] = i # get the last location of each char
8         last_loc = -1
9         prev_loc = -1
10        ans = []
11        for i, c in enumerate(S):
12            #prev_loc = min(prev_loc, i)
13            last_loc = max(last_loc, loc[c])
14            if i == last_loc: ##a good one
15                ans.append(last_loc - prev_loc)
16                prev_loc = last_loc
17
18        return ans

```

With the greedy approach, we further decreased the complexity to $O(n)$.

17.5.3 Data Compression, File Merge

File Merge Given array F of size n , each item indicates the length of the i -th file in the array. Find the best ordering of merging all files.

For example, $F = \{10, 5, 100, 50, 20, 15\}$

First, this is a exponential problem because we might need to try all permutation of a file. Merge the first two files take $10+5$ cost, and merge this further with 100 , takes $10+5+100$, and so. Now, let us write the cost of the original order:

$$c = \min(F_0 + F_1) + (F_0 + F_1 + F_2) + \dots + (F_0 + F_1 + F_2 + \dots + F_{n-1}) \quad (17.17)$$

$$= \min(n-1)F_0 + (n-2)F_1 + \dots + F_{n-1} \quad (17.18)$$

From the objective function, because all file size are having positive sizes, to minimize it, we have to make sure F_0 is the smallest item in the array because it has to be computed the most times, and F_2 is the second smallest and so on. We can easily figure out that sorting the files in increasing orders and merge them in this order result the least cost of merging. This is a very simple and natural greedy approach.

Data Compression

17.5.4 Fractional S**17.5.5 Graph Algorithms****17.6 Exercises**

- 630. Course Schedule III (hard)

18

Hands-on Algorithmic Problem Solving

The purpose of this chapter to see how our learned algorithm design principle can be applied into problem solving. We approach problems using different algorithm design principle and step by step to see how the change in the time and space complexity.

Longest Increasing Subsequence (300L) Given an unsorted array of integers, find the length of longest increasing subsequence.

Example :

Input : [10 ,9 ,2 ,5 ,3 ,7 ,101 ,18]

Output: 4

Explanation: The longest increasing subsequence is [2 ,3 ,7 ,101] , therefore the length is 4.

18.1 Direct Approach

18.1.1 Search in Graph

In a subsequence, an item can only have two choice: either in or out of the resulting subsequence, this makes our total subsequence $O(2^n)$. As we know the searching process shall always be a search tree, we now start to generate this subsequence, which starts from []. At the first level, we have item 10 that have two actions: not adding or adding, which makes it a two branches. At the second level, we consider item 9. This makes a search space of a binary tree, and we generate node implicitly since we only need to track the length of the path so far, and we need value of the last item to decide the children of current level. So far, we managed to model our

problem as finding the longest path in the search tree, which is a binary tree and with height n . We can have the Python code:

```

1 def lengthOfLIS(self, nums: List[int]) -> int:
2     def dfs(nums, idx, cur_len, last_num, ans):
3         if idx >= len(nums):
4             ans[0] = max(ans[0], cur_len)
5             return
6         if nums[idx] > last_num:
7             dfs(nums, idx+1, cur_len + 1, nums[idx], ans)
8             dfs(nums, idx+1, cur_len, last_num, ans)
9         ans = [0]
10        last_num = -sys.maxsize
11        dfs(nums, 0, 0, last_num, ans)
12        return ans[0]
```

18.1.2 Self-Reduction

Now, let us us an example smaller than before, say $[2, 5, 3, 7]$, which has the LIS 3 with $[2, 3, 7]$. Let us consider each state not atomic but as a subproblem. The same tree, but we translate each node differently. We start to consider the problem top down: we have problem $[2, 5, 3, 7]$, and our start index = 0, meaning start from item 2, then our problem is can be divided into different situations:

- not take 2: we find the LIS length of subproblem $[5, 3, 7]$. In this case, our subsequence can start from any of these 3 items, we indicate this case by not changing the previous value. Use `idx` to indicate the subproblem/subarray, we call `dfs` that `idx+1`.
- take 2: we need to find the LIS length of subproblem $[5, 3, 7]$ whose subsequence must start from 5. Thus, we set the `last_num` to 5 in the recursive call.

Therefore, our code becomes:

```

1 def lengthOfLIS(self, nums: List[int]) -> int:
2     def dfs(nums, idx, last_num):
3         if not nums:
4             return 0
5         if idx >= len(nums):
6             return 0
7         len1 = 0
8         if nums[idx] > last_num:
9             len1 = 1 + dfs(nums, idx+1, nums[idx])
10            len2 = dfs(nums, idx+1, last_num)
11            return max(len1, len2)
12
13        last_num = -sys.maxsize
14        return dfs(nums, 0, last_num)
```

In this solution, the time complexity has not improved yet, but from this approach, we can further increase the efficiency with dynamic programming.

18.1.3 Dynamic Programming

Memoization We have known that the recurrence relation takes $LIS(i, prev) = \max(LIS(i + 1, prev), LIS(i + 1, \text{nums}[i]))$. How many possible states for $LIS(i, prev)$? $i \in [0, n - 1]$, and $prev$ can have n candidates too, this makes the whole state space only n^2 . While, using the depth-first tree search we revisited a state multiple times, which eventually make the time complexity to $O(2^n)$. Now, let us modify the approach and use `memo` which is a dictionary and takes a tuple `(i, prev)` as key. If we found the state is not computed, we compute as we do in the previous implementation, if it exists in the memory, however, we just directly return the value and avoid recomputing again:

```

1 def lengthOfLIS(self, nums: List[int]) -> int:
2     def dfs(nums, idx, last_num, memo):
3         if idx >= len(nums):
4             return 0
5         if (idx, last_num) not in memo:
6             len1 = 0
7             if nums[idx] > last_num:
8                 len1 = 1 + dfs(nums, idx+1, nums[idx], memo)
9             len2 = dfs(nums, idx+1, last_num, memo)
10            memo[(idx, last_num)] = max(len1, len2)
11        return memo[(idx, last_num)]
12
13    last_num = -sys.maxsize
14    memo = {}
15    return dfs(nums, 0, last_num, memo)

```

18.2 A to B

Another approach is to use the concept of “prefix” or “suffix”. The LIS must start from one of the items in the array. Finding the length of the LIS in the original array can be achieved by comparing n subproblems, the length of LIS of:

```

1 [2, 5, 3, 7], LIS starts at 2,
2 [5, 3, 7], LIS starts at 5,
3 [3, 7], LIS starts at 3
4 [7], LIS starts at 7

```

18.2.1 Self-Reduction

We model the problem as in Fig. 29.1. Same here, our problem become finding the longest path in a N-ary tree instead of a binary tree. Define $f(i)$ as the LIS starting with index i in the array. then, its relation with other state will be $f(i) = \max_j(f(j)) + 1, j > i, a[j] > a[i]$, and $f[n] = 0$. Here, the base case is when there has element to start from which will have 0 LIS.

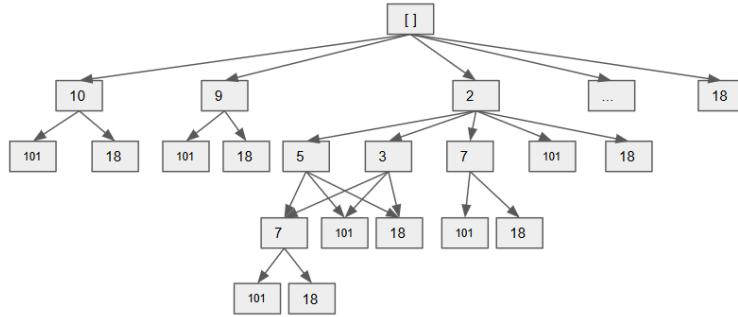


Figure 18.1: Graph Model for LIS, each path represents a possible solution.

```

1 def lengthOfLIS(self, nums: List[int]) -> int:
2     def dfs(nums, idx, cur_num):
3         max_len = 0
4         # Generate the next node
5         for i in range(idx+1, len(nums)):
6             if nums[i] > cur_num:
7                 max_len = max(max_len, 1 + dfs(nums, i, nums[i]))
8
9     return max_len
10    return dfs(nums, -1, -sys.maxsize)

```

18.2.2 Dynamic Programming

Memoization Similar to the last approach, we can write code:

```

1 def lengthOfLIS(self, nums: List[int]) -> int:
2     def dfs(nums, idx, cur_num, memo):
3         max_len = 0
4         # Generate the next node
5         if (idx, cur_num) not in memo:
6             for i in range(idx+1, len(nums)):
7                 if nums[i] > cur_num:
8                     max_len = max(max_len, 1 + dfs(nums, i, nums[i], memo))
9                     memo[(idx, cur_num)] = max_len
10        return memo[(idx, cur_num)]
11    memo = {}
12    return dfs(nums, -1, -sys.maxsize, memo)

```

Tabulation With the bottom-up manner, we need to tweet our above recurrence function and definition of state. The subproblem $f(i)$ here will be defined as the LIS ending at index i . We shall pay attention that with n elements there should exist $n+1$ states in total, that there is an empty state with empty array $[]$. The recurrence function will be shown in Eq. 18.1. It can be explained the LIS ending at index i will be transitioned from LIS

ending at any previous index by plusing one. The whole analysis process is illustrated in Fig 18.2.

$$f(i) = \begin{cases} 1 + \max(f(j)), & -1 \leq j < i < n, arr[j] < arr[i]; \\ 0 & \text{otherwise} \end{cases} \quad (18.1)$$

To simply the implementation, we insert a $-\infty$ value at the beginning of

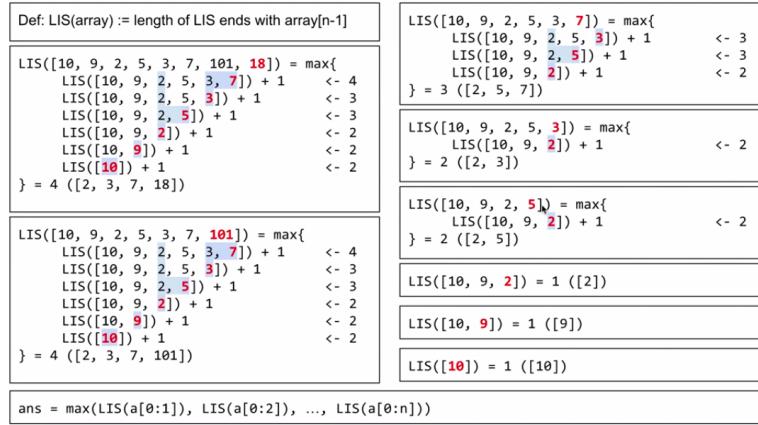


Figure 18.2: The solution to LIS.

the array. To initialize we set $dp[0] = 0$, and the answer is $\max(dp)$. The time complexity is $O(n^2)$ because we need two for loops: one outsider loop with i , and another inside loop with j . The space complexity is $O(n)$. The Python code is:

```

1 def lis(a):
2     # define the dp array
3     dp = [0] * (len(a)+1)
4     a = [-sys.maxsize] + a
5     print(a)
6     for i in range(len(a)): # end with index i-1
7         for j in range(i):
8             if a[j] < a[i]:
9                 dp[i] = max(dp[i], dp[j] + 1)
10    print(dp)
11    return max(dp)

```

18.2.3 Divide and Conquer

We can even speedup further by using binary search, the second loop we can use a binary search to make the time complexity $O(\log n)$, and the dp array used to save the maximum ans. Each time we use binary search to find an insertion point, if it is at the end, then the length grow. [4]->[4,10],->[4,10],[3,10],->[3,8]->[3,8,9]

```

1 def lengthOfLIS(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: int
5     """
6     def binarySearch(arr, l, r, num):
7         while l < r:
8             mid = l + (r - 1) // 2
9             if num > arr[mid]:
10                 l = mid + 1
11             elif num < arr[mid]:
12                 r = mid
13             else:
14                 return mid
15     return l
16     max_count = 0
17     if not nums:
18         return 0
19     dp = [0 for _ in range(len(nums))] #save the maximum till
now
20     maxans = 1
21     length = 0
22     for idx in range(0, len(nums)): #current combine this to
this subsequence, 10 to [], 9 to [10]
23         pos = binarySearch(dp, 0, length, nums[idx]) #find
insertion point
24         dp[pos] = nums[idx] #however if it is not at end, we
replace it, current number
25         if pos == length:
26             length += 1
27     print(dp)
28     return length

```

```

1 def lengthOfLIS(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: int
5     """
6     def binarySearch(arr, l, r, num):
7         while l < r:
8             mid = l + (r - 1) // 2
9             if num > arr[mid]:
10                 l = mid + 1
11             elif num < arr[mid]:
12                 r = mid
13             else:
14                 return mid
15     return l
16     max_count = 0
17     if not nums:
18         return 0
19     dp = [0 for _ in range(len(nums))]
20     LIS = 0
21     for i in range(len(nums)): #current combine this to this

```

```
22     subsequence , 10 to [] , 9 to [10]
23         pos = binarySearch (dp,0,length ,nums[ i ])
24         dp[ pos]= nums[ i ]
25         if pos==LIS:
26             LIS += 1
27     return LIS
```


Part V

Classical Algorithms

In this part, we focus on application through solving a few families of classical real-problems, ranging from advanced search algorithms on linear data structures, advanced graph algorithms, to typical string pattern matching. By studying and analyzing each problem's representative algorithm whereby the fundamental algorithm design and analysis principles are leveraged, we further enforce our skills to algorithmic problem solving.

Advanced Search on Linear Data Structures

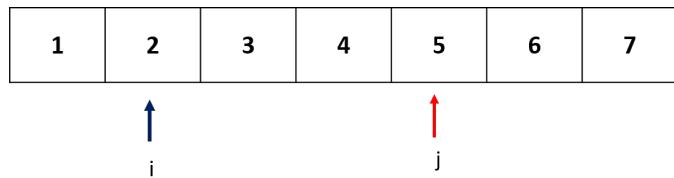


Figure 19.1: Two pointer Technique

On linear data structures, or on implicit linear state space, either a particular targeted item or a consecutive substructure such as a subarray and substring can be searched.

To find a single item on linear space, we can apply linear search in general, or binary search if the data structure is ordered/sorted with logarithmic cost. In this chapter, we introduce two pointer techniques that are commonly used to solve two types of problems:

1. Searching: To search for an item such as median, a predefined substructure, and a substructure that satisfy certain conditions such as finding the minimum subarray length wherein the subarray equals to a targeted sum. Or find a substructure satisfy a string pattern.
2. Adjusting: To adjust ordering or arrangement of items in the data structure such as removing duplicates from sorted array.

As the name suggests, Two pointers technique involves two pointers that start and move with the following two patterns:

1. Equi-directional: Both pointers start from the beginning of the array, and usually one moves faster and the other slower. Sliding window algorithm can be put into this category.
2. Opposite-directional: One pointer starts at the start position and conversely the other pointer starts at the end. These two oppositely posed pointers move toward each other and usually meet in the middle.

In the following sections, we will detail on two-pointer technique exemplified on real interview questions.

19.1 Slow-Faster Pointers

Suppose we have two pointers, i and j , which may or may not start at the start position in the linear data structures, but one moves slower (i) and the other faster (j). Two pointers can decide either a pair or a subarray to solve related problems. For the case of subarray, the algorithm is called sliding window algorithm. On the span of the array, and at most of three potential sub-spaces exist: from start index to i ($[0, i]$), from i to j ($[i, j]$), and from j to the end index ($[j, n]$).

Even though slow-faster pointers technique rarely given formal introduction in book, it is widely used in algorithms. In sorting, Lumuto's partition in the QuickSort used the slow-faster pointers to divide the whole region into three parts according the comparison result to the pivot: Smaller Items region, Larger Items region, and the unrestricted region. In string pattern matching, fixed sliding window and one we will introduce in this chapter.

In this section, we explain how two pointers work on two types of linear data structures: Array and Linked List.

19.1.1 Array

Remove Duplicates from Sorted Array(L26)

Given a sorted array $a = [0, 0, 1, 1, 2, 2, 3, 3, 4]$, remove the duplicates in-place such that each element appears only once and return the new length. Do not allocate extra space for another array, you must do this by modifying the input array in-place with $O(1)$ extra memory. In the given example, there are in total of 5 unique items and 5 is returned.

Analysis We set both slower pointer i and the faster pointer j at the first item in the array. Recall that slow-fast pointers cut the space of the sorted array into three parts, we can define them as:

1. unique items in region $[0, i]$,
2. untouched items in region $[i + 1, j]$,

3. and unprocessed items in region $[j + 1, n]$.

In the process, we compare the items pointed by two pointers, once these two items does not equal, we find an new unique item. We copy this unique item at the faster pointer right next to the position of the slower pointer. Afterwards, we move the slow pointer by one position to remove duplicates of our copied value.

With our example, at first, $i = j = 0$, region one has one item which is naively unique and region two has zero item. Part of the process is illustrated as:

i	j	$[0, i]$	$[i+1, j]$	process
0	0	$[0]$	$[]$	item $0==0, j+1=1$
0	1	$[0]$	$[0]$	item $0==0, j+1=2$
0	2	$[0]$	$[0, 1]$	item $0!=1, i+1=1, \text{copy } 1 \text{ to index } 1, j+1=3$
1	3	$[0, 1]$	$[1, 1]$	item $1==1, j+1=4$
1	4	$[0, 1]$	$[1, 1, 1]$	item $1==1, j+1=5$
1	5	$[0, 1]$	$[1, 1, 1, 2]$	item $1==2, i+1=2, \text{copy } 2 \text{ to index } 2, j+1=6$
2	6	$[0, 1, 2]$	$[1, 1, 2, 2]$	

The code is given as:

```

1 def removeDuplicates(nums) -> int:
2     i, j = 0, 0
3     while j < len(nums):
4         if nums[i] != nums[j]:
5             # Copy j to i+1
6             i += 1
7             nums[i] = nums[j]
8         j += 1
9     return i + 1

```

After calling the above function on our given example, array a becomes $[[0, 1, 2, 3, 4, 2, 2, 3, 3, 4]]$. Check the source code for the whole visualized process.

Minimum Size Subarray Sum(L209)

Given an array of n positive integers and a positive integer s , find the minimal length of a contiguous subarray of which the sum $\geq s$. If there isn't one, return 0 instead.

Example :

Input: $s = 7$, $\text{nums} = [1, 4, 1, 2, 4, 3]$
Output: 2

Explanation: the subarray $[4, 3]$ has the minimal length under the problem constraint .

Analysis In this problem, we need to secure a substructure—subarray—that not only satisfies a condition($sum \geq s$) but also has the minimal length. Naively, we can enumerate all subarrays and search through them to find the minimal length, which requires at least $O(n^2)$ time complexity using prefix sum. The code is as:

However, we can use two pointers i and j ($i \leq j$) and both points at the first item. In this case, these two pointers defines a subarray $a[i : j + 1]$ and we care the region $[i, j]$. As we increase pointer j , we keep adding positive item into the sum of the subarray, making the subarray sum monotonically increasing. Oppositely, if we increase pointer i , we remove positive item away from the subarray, making the sum of the subarray monotonically decreasing. The detailed steps of two pointer technique in this case is as:

1. Get the optimal subarray for all subproblems(subarries) that start from current i , which is 0 at first. We accomplish this by forwarding j pointer to include enough items until $sum \geq s$ that we pause and go to the next step. Let's assume pointer j stops at e_0 .
2. Get the optimal subarray for all subproblems(subarries) that end with current j , which is e_0 at the moment. We do this by forwarding pointer i this time to shrink the window size until $sum \geq s$ no longer holds. Let's assume pointer i stops at index s_0 . Now, we find the optimal solution for subproblems $a[0 : i, 0 : j]$ (denoting subarries with the start point in range $[0, i]$ and the end point in range $[0, j]$).
3. Now that $i = s_0$ and $j = e_0$, we repeat step 1 and 2.

In our example, we first move j until $j = 3$ with a subarray sum of 8. Then we move pointer i until $i = 1$ when the subarray sum is less than 7. For subarray $[1, 4, 1, 2]$, we find its optimal solution to have a length 3. The Python code is given as:

```

1 def minSubArrayLen(s: int, nums) -> int:
2     i, j = 0, 0
3     acc = 0
4     ans = float('inf')
5     while j < len(nums):
6         acc += nums[j]
7         # Shrink the window
8         while acc >= s:
9             ans = min(ans, j - i + 1)
10            acc -= nums[i]
11            i += 1
12        j += 1
13
14    return ans if ans < float('inf') else 0

```

Because both pointer i and j move at most n steps, with the total operations to be at most $2n$, making the time complexity as $O(n)$. The above question would be trivial if the maximum subarray length is asked.

19.1.2 Minimum Window Substring (L76, hard)

Given a string S and a string T , find all the minimum windows in S which will contain all the characters in T in complexity $O(n)$.

Example :

Input : $S = "AOBECDBANC"$, $T = "ABC"$

Output : ["CDBA", "BANC"]

'A'	'B'	'C'	count
1	1	1	3

Figure 19.2: The data structures to track the state of window.

Analysis Applying two pointers, with the region between pointer i and j to be our testing substring. For this problem, the condition for the window $[i, j]$ it will at most have all characters from T . The intuition is we keep expanding the window by moving forward j until all characters in T is found. Afterwards, we contract the window so that we can find the minimum window with the condition satisfied. Instead of using another data structure to track the state of the current window, we can depict the pattern T as a dictionary data structure where all unique characters comprising the keys and with the number of occurrence of each character as value. We use another variable **count** to track how the number of unique characters. In all, they are used to track the state of the moving window in $[i, j]$, with the value of the dictionary to indicate how many occurrence is short of, and the **count** represents how many unique characters is not fully found, and we depict the state in Fig. 19.2.

Along the expanding and shrinking of the window that comes with the movement of pointer i and j , we track the state with:

- When forwarding j , we encompass $S[j]$ in the window. If $S[j]$ is a key in the dictionary, decrease the value by one. Further, if the value reaches to the threshold 0, we decrease **count** by one, meaning we are short of one less character in the window.

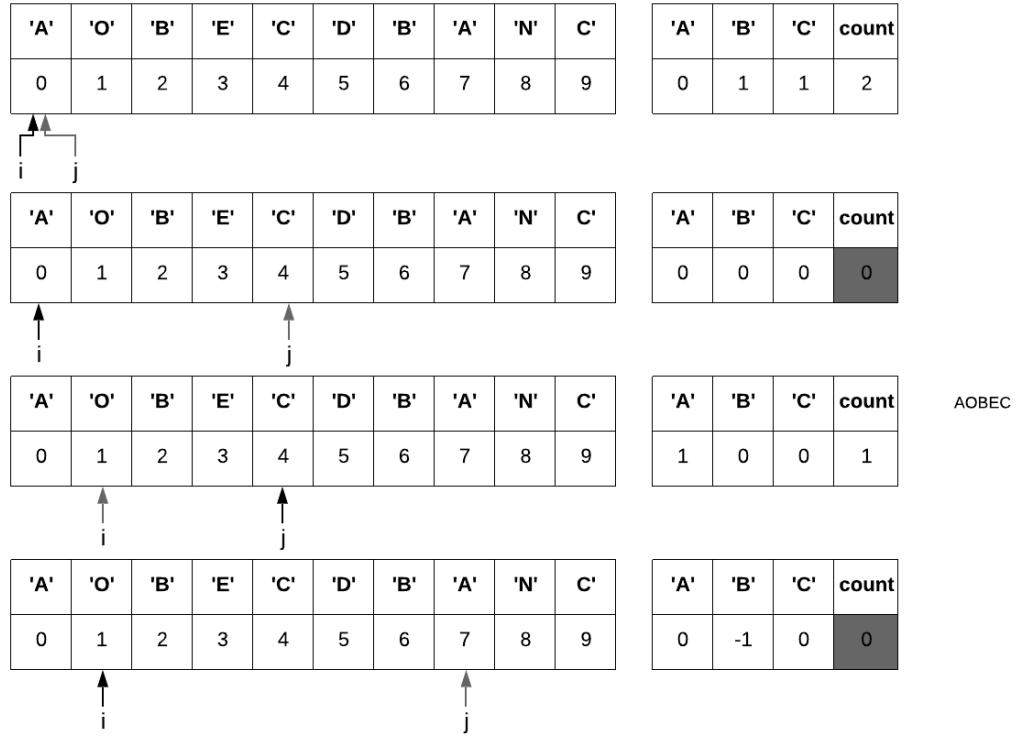


Figure 19.3: The partial process of applying two pointers. The grey shaded arrow indicates the pointer that is on move.

- Once `count=0`, our window satisfy condition for contracting. We then forward i , removing $S[i]$ from the window if it is existing key in the dictionary by increasing this key's value, meaning the window is short of one more character. Once the value reaches to the threshold of 1, we increase `count`.

Part of this process with our example is shown in Fig. 19.3. And the Python code is given as:

```

1 from collections import Counter
2 def minWindow(s, t):
3     dict_t = Counter(t)
4     count = len(dict_t)
5     i, j = 0, 0
6     ans = []
7     minLen = float('inf')
8     while j < len(s):
9         c = s[j]
10        if c in dict_t:
11            dict_t[c] -= 1
12            if dict_t[c] == 0:

```

```

13     count == 1
14 # Shrink the window
15 while count == 0 and i < j:
16     curLen = j - i + 1
17     if curLen < minLen:
18         minLen = j - i + 1
19         ans = [s[i:j+1]]
20     elif curLen == minLen:
21         ans.append(s[i:j+1])
22
23     c = s[i]
24     if c in dict_t:
25         dict_t[c] += 1
26         if dict_t[c] == 1:
27             count += 1
28     i += 1
29
30     j += 1
31 return ans

```

19.1.3 When Two Pointers do not work

Two pointer does not always work on subarray related problems.

 **What happens if there exists negative number in the array?**

Since the sum of the subarray is no longer monotonically increasing with the number of items between two pointers, we can not figure out how to move two pointers each step. Instead (1) we can use prefix sum and organize them in order, and use binary search to find all possible start index. (2) use monotone stack (see LeetCode probelm: 325. Maximum Size Subarray Sum Equals k, 325. Maximum Size Subarray Sum Equals k (hard))

 **What if we are to check the maximum average subarray?**

644. Maximum Average Subarray II (hard). Similarly, the average of subarray does not follow a certain order with the moving of two pointers at each side, making it impossible to decide how to make the two pointers.

19.1.4 Linked List

The complete code to remove cycle is provided in google colab together with running examples.

Middle of the Linked List(L876)

Given a non-empty, singly linked list with head node *head*, return a middle node of linked list. When the linked list is of odd length, there exists one and only middle node, but when it is of even length, two exists and we return the second middle node.

Example 1 (odd length) :

```
Input : [1 ,2 ,3 ,4 ,5]
Output: Node 3 from this list (Serialization : [3 ,4 ,5])
```

Example 2 (even length) :

```
Input : [1 ,2 ,3 ,4 ,5 ,6]
Output: Node 4
```

```
from this list (Serialization : [4 ,5 ,6])
```

Analysis If the data structure is array, we can compute the position of the middle item simply with the total length. Following this method, if only one pointer is applied, we can first iterate over the whole linked list in $O(n)$ time to get the length. Then we do another iteration to obtain the middle node. $n + \frac{n}{2}$ times of operations needed, making the time complexity $O(n)$.

However, we can apply two pointers simultaneously at the head node, each one moves at different paces: the slow pointer moves one step at a time and the fast moves two steps instead. When the fast pointer reached the end, the slow pointer will stop at the middle. This slow-faster pointers technique requires only $\frac{n}{2}$ times of operations, which is three times faster than our naive method, although the big Oh time complexity still remains $O(n)$.

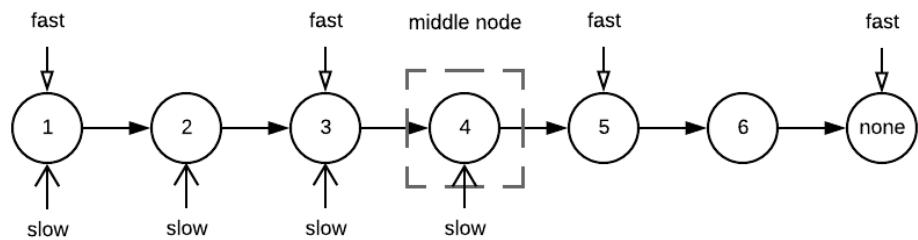


Figure 19.4: Slow-fast pointer to find middle

Implementation Simply, we illustrate the process of running the two pointers technique on our two examples in Fig. 19.4. As we can see, when the slow pointer reaches to item 3, the faster pointer is at item 5, which

is the last item in the first example that comes with odd length. Further, when the slow pointer reaches to item 4, the faster pointer reaches to the empty node of the last item in the second example that comes with even length. Therefore, in the implementation, we check two conditions in the `while` loop:

1. For example 1: if the fast pointer has no successor (`fast.next==None`), the loop terminates.
2. For example 1: if the fast pointer is invalid (`fast==None`), the loop terminates.

The Python code is as:

```

1 def middleNode(head):
2     slow = fast = head
3     while fast and fast.next:
4         fast = fast.next.next
5         slow = slow.next
6     return slow

```

Floyd's Cycle Detection (Floyd's Tortoise and Hare)

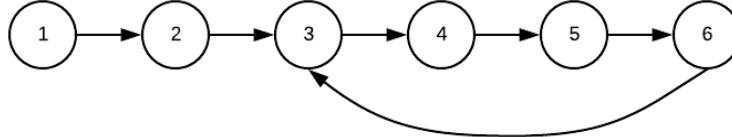


Figure 19.5: Circular Linked List

When a linked list which has a cycle, as shown in Fig. 19.5, iterating items over the list will make the program stuck into infinite loop. The pointer starts from the heap, traverse to the start of the loop, and then comes back to the start of the loop again and continues this process endlessly. To avoid being stuck into a “trap”, we have to possibly solve the following three problems:

1. Check if there exists a cycle.
2. Check where the cycle starts.
3. Remove the cycle once it is detected.

The solution encompasses the exact way of slow faster pointers traversing through the linked list as our last example. With the slow pointer iterating one item at a time, and the faster pointer in double pace, these two pointers will definitely meet at one item in the loop. In our example, they will meet

at node 6. So, is it possible that it will meet at the non-loop region starts from the heap and ends at the start node of the loop? The answer is No, because the faster pointer will only traverse through the non-loop region once and it is always faster than the slow pointer, making it impossible to meet in this region. This method is called Floyd's Cycle Detection, aka Floyd's Tortoise and Hare Cycle Detection. Let's see more details at how to solve our mentioned three problems with this method.

Check Linked List Cycle(L141) Compared with the code in the last example, we only need to check if the `slow` and `fast` pointers are pointing at the same node: If it is, we are certain that there must be a loop in the list and return `True`, otherwise return `False`.

```

1 def hasCycle(head):
2     slow = fast = head
3     while fast and fast.next:
4         slow = slow.next
5         fast = fast.next.next
6         if slow == fast:
7             return True
8     return False

```

Check Start Node of Linked List Cycle(L142) Given a linked list, return the node where the cycle begins. If there is no cycle, return `None`.

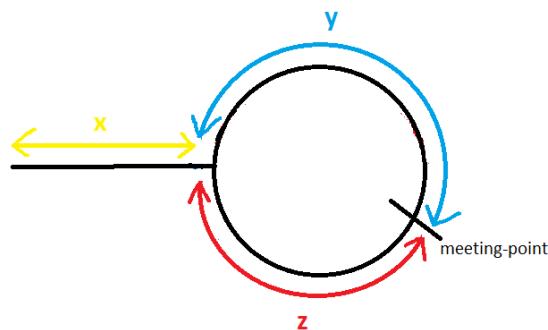


Figure 19.6: Floyd's Cycle finding Algorithm

For a given linked list, assume the slow and fast pointers meet at node somewhere in the cycle. As shown in Fig. 19.6, we denote three nodes: head (h , start node of cycle(s), and meeting node in the cycle(m). we denote the distance between h and s to be x , the distance between s and m to be y , and the distance between m and s to be z . Because the faster pointer traverses through the list in double speed, when it meets up with the slow pointer, the distance that it traveled($x + y + z + y$) to be two times of the distance

traveled by the slow pointer ($x + y$).

$$x + y + z + y = x + y \quad (19.1)$$

$$x = z \quad (19.2)$$

From the above equation, we obtain the equal relation between x and z . the starting node of the cycle from the head is x , and y is the distance from the start node to the slow and fast pointer's node, and z is the remaining distance from the meeting point to the start node. Therefore, after we have detected the cycle from the last example, we can reset the slow pointer to the head of the linked list after. Then we make the slow and the fast pointer both traverse at the same pace—one node at a time—until they meet at a node we stop the traversal. The node where they stop at is the start node of the cycle. The code is given as:

```

1 def detectCycle(head):
2     slow = fast = head
3
4     def getStartNode(slow, fast, head):
5         # Reset slow pointer
6         slow = head
7         while fast and slow != fast:
8             slow = slow.next
9             fast = fast.next
10        return slow
11
12    while fast and fast.next:
13        slow = slow.next
14        fast = fast.next.next
15        # A cycle is detected
16        if slow == fast:
17            return getStartNode(slow, fast, head)
18
19    return None

```

Remove Linked List Cycle We can remove the cycle by recirculating the last node in the cycle, which in example in Fig. 19.5 is node 6 to an empty node. Therefore, we have to modify the above code to make the `slow` and `fast` pointers stop at the last node instead of the start node of the loop. This subroutine is implemented as:

```

1 def resetLastNode(slow, fast, head):
2     slow = head
3     while fast and slow.next != fast.next:
4         slow = slow.next
5         fast = fast.next
6         fast.next = None

```

The complete code to remove cycle is provided in google colab together with running examples.



What if there has not only one, but multiple cycles in the Linked List?

19.2 Opposite-directional Pointers

Another variant of two pointers technique is to place these two pointers oppositely: one at the beginning and the other at the end of the array. Through the process, they move toward each other until they meet in the middle. Details such as how much each pointer moves or which pointer to move at each step decided by our specific problems to solve. We just have to make sure when we are applying this technique, we have considered its whole state space, and will not miss out some area which makes the search incomplete.

The simplest example of this two pointers method is to reverse an array or a string around. For example, when the list $a = [1, 2, 3, 4, 5]$ is reversed, it becomes $[5, 4, 3, 2, 1]$. Of course we can simply assign a new list and copy the items in reversed orders. But, with two pointers, we are able to reverse it in-place and using only $O(\frac{n}{2})$ times of operations through the following code:

```

1 def reverse(a):
2     i, j = 0, len(a) - 1
3     while i < j:
4         # Swap items
5         a[i], a[j] = a[j], a[i]
6         i += 1
7         j -= 1

```

Moreover, binary search can be viewed as an example of opposite-directional pointers. At first, these two pointers are the first and the last item in the array. Then depends on which side of the target compared with the item in the middle, one of the pointers move either forward or backward to the middle point, reducing the search space to half of where it started at each step. We also explore another example with this technique.

Two Sum on Sorted Array(L167)

Given an array of integers that is already sorted in ascending order, find two numbers such that they add up to a specific target number.

Input: numbers = [2, 7, 11, 15], target = 9

Output: [1, 2]

Explanation: The sum of 2 and 7 is 9. Therefore index1 = 0, index2 = 1.

Analysis If we simply put enumerate all possible pairs, we have to take $O(n^2)$ to solve this problem. However, with the opposite-directional two pointers, it gives out linear performance.

Denote the list as $A = [a_1, a_2, \dots, a_{n-1}, a_n]$, and for the sorted array we have $a_1 \leq a_2 \leq \dots \leq a_{n-1} \leq a_n$. The range of the sum of any two items in the array is within two possible ranges: $[a_1 + a_2, a_1 + a_n]$ and $[a_1 + a_n, a_{n-1} + a_n]$. By placing one pointer i at a_1 and the other j at a_n to start with, we can get $a_1 + a_n$ as the sum. Pointer i can only move forward, accessing larger items. On the other hand, pointer j can only backward, accessing smaller items. Now there are three scenarios according to the comparison between the target and the current sum of the two pointers:

1. If $t == a[i] + a[j]$, target sum found.
2. If $t > a[i] + a[j]$, we have to increase the sum, we can only do this by moving pointer i forward.
3. If $t < a[i] + a[j]$, we have to decrease the sum, we can only do this by moving pointer j backward.

The Python code is as:

```

1 def twoSum(a, target):
2     n = len(a)
3     i, j = 0, n-1
4     while i < j:
5         temp = a[i] + a[j]
6         if temp == target:
7             return [i, j]
8         elif temp < target:
9             i += 1
10        else:
11            j -= 1
12    return []

```

19.3 Follow Up: Three Pointers

Sometimes, manipulating two pointers is not even enough to distinguish different subspaces, we might need to the assistant of one another pointer to make things work.

Binary Subarrays With Sum (L930)

In an array A of 0s and 1s, how many non-empty subarrays have sum S ?

Example 1:

Input: $A = [1, 0, 1, 0, 1]$, $S = 2$

Output: 4

Explanation :

The 4 subarrays are listed below:

```
[1,0,1], index (0, 2)
[1,0,1,0], index (0, 3)
[0,1,0,1], index (1, 4)
[1,0,1], index (2, 4)
```

Analysis This problem is highly similar to the minimum length subarray problem we encountered before. We naturally start with two pointers i and j , and restrict the subarray in range $[i, j]$ to satisfy condition $\text{sum} \leq S$. The window is contracted when the condition is violated. We would have write the following code:

```
1 def numSubarraysWithSum(a, S):
2     i, j = 0, 0
3     win_sum = 0
4     ans = 0
5     while j < len(a):
6         win_sum += a[j]
7         while i < j and win_sum > S:
8             win_sum -= a[i]
9             i += 1
10        if win_sum == S:
11            ans += 1
12            print('({}, {})'.format(i, j))
13        j += 1
14    return ans
```

However, the above code only returns 3, instead of 4 as shown in the example. By printing out pointers i and j , we can see the above code is missing case (2, 4). Why? Because we are restricting the subarray sum in range $[i, j]$ to be smaller than or equal to S , with the occurrence of 0s that might appear in the front or in the rear of the subarray:

- In the process of expanding the subarray, pointer j is moved one at a time. Thus, even though 0s appear in the rear of the subarray, the counting is correct.
- However, in the process of shrinking the subarray while the restriction is violated ($\text{sum} > S$), we stop right away once $\text{sum} \leq S$. And in the code, we end up only counting it as one occurrence. With 0s at the beginning of the subarray, such as the subarray [0, 1, 0, 1] with index 1 and 4, there count should be two instead of one.

The solution is to add another pointer i_h to handle the missed case: When the $\text{sum} = S$, count the total occurrence of 0 in the front. Compared with the above solution, the code only differs slightly with the additional pointer and one extra `while` loop to deal the case. Also we need to pay attention that $i_h \leq j$, otherwise, the `while` loop would fail with example with only zeros and a targeting sum 0.

```

1 def numSubarraysWithSum(a, S):
2     i, i_h, j = 0, 0, 0
3     win_sum = 0
4     ans = 0
5     while j < len(a):
6         win_sum += a[j]
7         while i < j and win_sum > S:
8             win_sum -= a[i]
9             i += 1
10    # Move i_h to count all zeros in the front
11    i_h = i
12    while i_h < j and win_sum == S and a[i_h] == 0:
13        ans += 1
14        i_h += 1
15
16    if win_sum == S:
17        ans += 1
18    j += 1
19    return ans

```

We noticed that in this case, we have to explicitly restrict $i < j$ and $i_h < j$ due to the special case, while in all our previous examples, we do not have to.

19.4 Summary

Two pointers is a powerful tool for solving problems on liner data structures, such as “certain” subarray and substring problems as we have shown in the examples. The “window” secluded between the two pointers can be viewed as sliding window: It can move slide forward with the forwarding the slower pointer. Two important properties are generally required for this technique to work:

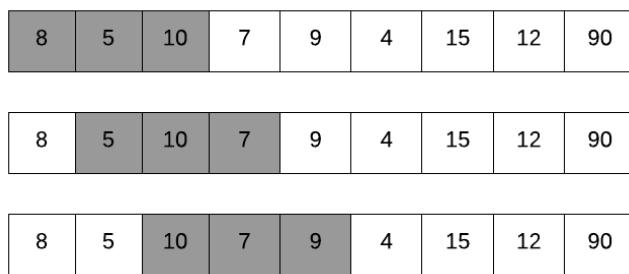


Figure 19.7: Sliding Window Property

1. Sliding window property: Either we move the faster pointer j forward by one, or move the slower pointer i , we can get the state of current window in $O(1)$ cost knowing the state of the last window.

For example, given an array, imagine that we have a fixed size window as shown in Fig. 19.7, and we can slide it forward one position at a time, compute the sum of each window. The bruteforce solution would be of $O(kn)$ complexity where k is the window size and n is the array size by using two nested `for` loops: one to set the starting point, and the other to compute the sum in $O(k)$. However, the sum of the current window (S_c) can be computed from the last window (S_l), and the items that just slid out and in as a_j and a_i respectively. Then $S_c = S_l - a_i + a_j$. Getting the state of the window between two pointers in $O(1)$ as shown in the example is our called Sliding Window Property.

Usually, for an array with numerical value, it satisfies the sliding window property if we are to compute its sum or product. For substring, as shown in our minimum window substring example, we can get the state of current window referring to the state of the last window in $O(1)$ with the assist of dictionary data structure. In substring, this is more obscure, and the general requirement is that the state of the substring does not relate to the order of the characters(anagram-like state).

2. Monotonicity: For subarray sum/product, the array should only comprise all positive/negative values so that the prefix sum/product has monotonicity: moving the faster pointer and the slower pointer forward results into opposite change to the state. The same goes for the substring problems where we see from the minimum window substring example the change of the state: `count` and the value of the dictionary is monotonic, and each either increases or decreases with the moving of two pointers.

19.5 Exercises

1. 3. Longest Substring Without Repeating Characters
2. 674. Longest Continuous Increasing Subsequence (easy)
3. 438. Find All Anagrams in a String
4. 30. Substring with Concatenation of All Words
5. 159. Longest Substring with At Most Two Distinct Characters
6. 567. Permutation in String
7. 340. Longest Substring with At Most K Distinct Characters
8. 424. Longest Repeating Character Replacement

20

Advanced Graph Algorithms

Our standing at graph algorithms:

1. Search Strategies (Chapter)
2. Combinatorial Search(Chapter)
3. Advanced Graph Algorithm(Current)
4. Graph Problem Patterns(Future Chapter)

This chapter is more to apply the basic search strategies and two advanced algorithm design methodologies—Dynamic Programming and Greedy Algorithms—on a variety of classical graph problems:

- Cycle Detection (Section 20.1), Topological Sort(Section 20.2), and Connected Components(Section 20.3) which all require a thorough understanding to properties of basic graph search, especially Depth-first graph search.
- On the other hand, Minimum Spanning Tree (MST) and Shortest Path Algorithm on the entails our mastering of Breath-first Graph Search.
- Moreover, to achieve better efficiency, Dynamic Programming and Greedy Algorithms has to be leveraged in the graph search process. For example, Bellman-Ford algorithm uses the Dynamic Programming to avoid recomputing intermediate paths while searching the shortest paths from a single source to all other targets. The classical Prim's and Kruskal's MST algorithm both demonstrates how greedy algorithm can be applied, each in a different way.

20.1 Cycle Detection

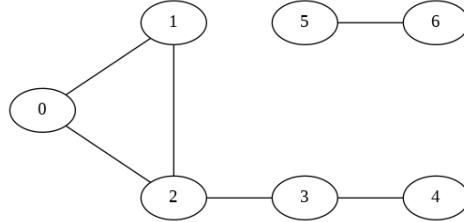


Figure 20.1: Undirected Cyclic Graph. $(0, 1, 2, 0)$ is a cycle

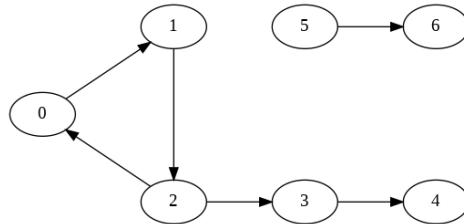


Figure 20.2: Directed Cyclic Graph, $(0, 1, 2, 0)$ is a cycle.

Problem Definition Detect cycles in both directed and undirected graph. Specifically, given a path with $k+1$ vertices, denoted as v_0, v_1, \dots, v_k in graph G :

1. When G is directed: a cycle is formed if $v_0 = v_k$ and the path contains at least one edge. For example, there is a cycle $0, 1, 2, 0$ shown in the directed graph of Fig. 20.2.
2. When G is undirected: the path forms a cycle only if $v_0 = v_k$ and the path length is at least three (i.e., there are at least three distinct vertices within the path). For example, in the undirected graph of Fig. 20.2, we couldn't say $(0, 2)$ is a cycle even though there is a path $0, 2, 0$, but the path $0, 2, 1, 0$ is as the path length ≥ 3 .

DFS to Solve Cycle Detection Recall the process of DFS graph search where a vertex has three possible states—white, gray, and black. A back edge appears while we reach to an adjacent vertex v which is in gray state from current vertex u . If we connect v back to its ancestor u , we find our cycle if the graph is directed. When the graph is undirected, we have discussed that it has only tree edge and back edge. Thus, we will use two states: visited and not visited. For edge (u, v) , we check two conditions:

1. if v is visited already. In Fig. 20.1, when we are at 1, we first visit 0

2. avoiding cycle of length one which is any existing edge within the graph. We can easily achieve this by tracking the predecessor p of the exploring vertex during the search, and making sure the predecessor is not the same as the current vertex: $p \neq u$.

Cycle Detection for Directed Graph We define a function `hasCycleDirected` with g as the adjacent list of graph, $state$ as a list to track state for each vertex, and s as the exploring vertex. The function returns a boolean value to indicate if there is a cycle or not. The function is essentially a DFS graph search along with an extra condition check on the back edge.

```

1 def hasCycleDirected(g, s, state):
2     state[s] = STATE.gray # first be visited
3     for v in g[s]:
4         if state[v] == STATE.white:
5             if hasCycleDirected(g, v, state):
6                 print(f'Cycle found at node {v}.')
7                 return True
8         elif state[v] == STATE.gray: # a back edge
9             print(f'Cycle starts at node {v}.')
10            return True
11     else:
12         pass
13     state[s] = STATE.black # mark it as complete
14

```

Because a graph can be disconnected with multiple components, we run `hasCycleDirected` on each unvisited vertex within the graph in a main function.

```

1 def cycleDetectDirected(g):
2     n = len(g)
3     state = [STATE.white] * n
4     for i in range(n):
5         if state[i] == STATE.white:
6             if hasCycleDirected(g, i, state):
7                 return True
8     return False

```

Cycle Detection for Undirected Graph First, we add another variable p to track the predecessor. p will first be initialized to -1 because the root in the rooted search tree has no predecessor (or ancestor). We can use the three coloring state as we did in directed graph, but it is a slight overkill. In the implementation, we only use boolean value to mark its state:

```

1 def hasCycleUndirected(g, s, p, visited):
2     visited[s] = True
3     for v in g[s]:
4         if not visited[v]:
5             if hasCycleUndirected(g, v, s, visited):
6                 print(f'Cycle found at node {v}.')

```

```

7         return True
8     else:
9         if v != p: # both black and gray
10            print(f'Cycle starts at node {v}.')
11            return True
12     return False

```

The main function:

```

1 def cycleDetectUndirected(g):
2     n = len(g)
3     visited = [False] * n
4     for i in range(n):
5         if not visited[i]:
6             if hasCycleUndirected(g, i, -1, visited):
7                 print(f'Cycle found at start node {i}.')
8                 return True
9
10    return False

```

Please check the source code to try out the examples.



How to find all cycles? First, we need to enumerate all paths while searching in order to get all cycles. This requires us to retreat to less efficient search strategy: depth-first tree search. Second, for each path, we find where the cycle starts by comparing each v_i with current vertex u : in directed graph, once $v_i == u$, the cycle is $v_i, v_{i+1}, \dots, v_k, v_i$; in undirected graph, the cycle is found only if the length of $v_i, \dots, v_k \geq 3$.

20.2 Topological Sort

Problem Definition In a given Directed Acyclic Graph (DAG) $G = (V, E)$, *topological sort/ordering* of is a linear ordering of the vertices V , such that for each edge $e \in E, e = (u, v)$, u comes before v . If a vertex represents a task to be completed and each directed edge denotes the order between two tasks, then topological sort is a way of linearly ordering a number of tasks in a completable sequence.

Every DAG has at least one topological ordering. For example, the topological ordering of Fig 20.3 can be $[0, 1, 3, (2, 4, 5), 6]$, where $(2, 4, 5)$ can be of any order, i.e., $(2, 4, 5), (2, 5, 4), (4, 2, 5), (4, 5, 2), (5, 2, 4), (5, 4, 2)$.

A topological ordering is only possible if there is no cycle existing in the graph. Thus, a cycle detection should be applied first when we are given a possible cyclic graph.

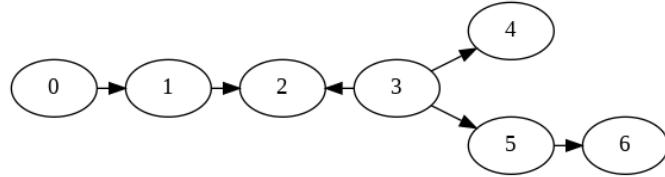


Figure 20.3: DAG 1

Kahn's algorithm (1962)

In topological sort, the first vertex is always ones with in-degree 0 (a vertex with no incoming edges). A naive algorithm is to decide the first node (with in-degree 0), add it in resulting order S , and remove all outgoing edges from this node. Repeat this process until:

- $V - S$ is empty, i.e., $|S| = |V|$, which indicates we found valid topological ordering.
- no node with 0 in-degree found in the remaining graph $G = (V - S, E')$ where E' are the remaining edges from E after the removal, i.e., $|S| < |V|$, indicating a cycle exists in $V - S$ and no valid answer exists.

For example, with the digraph in Fig. 20.3, the process is:

```

S      Removed Edges
0, 3 are the in-degree 0 nodes
Add 0      (0, 1)
1, 3 are the current in-degree 0 node
Add 1      (1, 2)
3 is the only in-degree 0 node
Add 3      (3, 2), (3, 4), (3,5)
2, 4, 5 are the in-degree 0 nodes
Add 2
Add 4
Add 5      (5, 6)
6 is the only in-degree 0 node
Add 6
V-S empty, stop
  
```

In this process, we see that in some time 2, 4, 5 are no in-degree 0 nodes, that is why their orderings can be permuted, resulting multiple topological orderings.

In implementation, instead of removing edges from the graph explicitly, a better option is to track $V - S$ with each vertex's in-degree: whenever a in-degree 0 vertex u is added into S , $\forall v, u \rightarrow v$, decrease the in-degree of v by one. We also keep a queue of the all nodes with in-degree zero Q . Whenever a vertex in $V - S$ is detected with zero in-degree, add it into Q . Accumulatively, the cost of decreasing the in-degree for vertices in $V - S$ is $|E|$ as from the start to end, “all edges are removed.” The cost of removing

of vertex from $V - S$ is $|V|$ as all nodes are removed at the end. With the initialization of the in-degree for vertices in $V - S$, we have a total of $O(2|E| + |V|)$, i.e., $O(|E| + |V|)$ as the time complexity. Python code:

```

1 from collections import defaultdict
2 import heapq
3 def kahns_topo_sort(g):
4     S = []
5     V_S = [(0, node) for node in range(len(g))] # initialize node
6         with 0 as in-degree
7     indegrees = defaultdict(int)
8     # Step 1: count the in-degree
9     for u in range(len(g)):
10         indegrees[u] = 0
11     for u in range(len(g)):
12         for v in g[u]:
13             indegrees[v] += 1
14     print(f'initial indegree : {indegrees}')
15     V_S = [(indegree, node) for node, indegree in indegrees.items()
16             ()]
17     heapq.heapify(V_S)
18
19     # Step 2: Kahn's algorithm
20     while len(V_S) > 0:
21         indegree, first_node = V_S.pop(0)
22         if indegree != 0: # cycle found, no topological ordering
23             return None
24         S.append(first_node)
25         # Remove edges
26         for v in g[first_node]:
27             indegrees[v] -= 1
28         # update V_S
29         for idx, (indegree, node) in enumerate(V_S):
30             if indegree != indegrees[node]:
31                 V_S[idx] = (indegrees[node], node)
32     heapq.heapify(V_S)
33
34     return S

```

Calling the function using graph in Fig. 20.3 gives result:

```

1 initial indegree : defaultdict(<class 'int'>, {0: 0, 1: 1, 2: 2,
2     3: 0, 4: 1, 5: 1, 6: 1})
3 [0, 1, 3, 2, 4, 5, 6]

```

Linear Topological Sort with Depth-first Graph Search

In depth-first graph search, if there is an edge $u \rightarrow v$, the recursive search from v will always be completed ahead of the search of u . With a simple reverse of the finishing ordering of vertices in depth-first graph search, the topological ordering takes $O(|E| + |V|)$ time. The time complexity equates to that of Kahn's algorithm, but this process is more efficient as it does not require the counting and updates of node in-degrees. The whole process

is exactly the same as Cycle Detection with additional complete ordering tracking.

First, the code of the DFS is:

```

1 def dfs(g, s, colors, complete_orders):
2     colors[s] = STATE.gray
3     for v in g[s]:
4         if colors[v] == STATE.white:
5             if dfs(g, v, colors, complete_orders):
6                 return True
7         elif colors[v] == STATE.gray: # a cycle appears
8             print(f'Cycle found at node {v}.')
9             return True
10    colors[s] = STATE.black
11    complete_orders.append(s)
12    return False

```

Then main function is:

```

1 def topo_sort(g):
2     n = len(g)
3     complete_orders = []
4     colors = [STATE.white] * n
5     for i in range(n): # run dfs on all the node
6         if colors[i] == STATE.white:
7             ans = dfs(g, i, colors, complete_orders)
8             if not ans:
9                 print('Cycle found, no topological ordering')
10            return None
11    return complete_orders[::-1]

```

Call `topo_sort` on the graph, we will have the sorted ordering as:

```

1 [3, 5, 6, 4, 0, 1, 2]

```

which is another linear topological ordering.

Example: Course Schedule (L210, m)

There are a total of n courses that you have to take. Some courses may have prerequisites, for example course 1 has to be taken before course 0, which is expressed as $[0, 1]$. Given the total number of courses and the prerequisite pairs, return the ordering of courses you should take to finish all courses. If it is impossible to finish, return an empty array.

Analysis Viewing a pair $[u, v]$ as an directed edge $v \rightarrow u$, we have a directed graph with n vertices and we solve the ordering of courses as getting the topological sort of vertices in the resulting digraph.

20.3 Connected Components

Problem Definition In graph theory, a *connected component*(or simply component) is defined as a subgraph where all vertices are mutually connected, i.e., where there exists a path between any two vertices in it. A graph $G = (V, E)$ is thus composed of separate connected components(sets) which are mutually exclusive and include all the vertices, .i.e., $V = V_0 \cup V_1 \cup \dots \cup V_{m-1}, V_i \cap V_j \neq \emptyset$. A connected component algorithm should be able to cluster vertices of each single connected component. For example, the

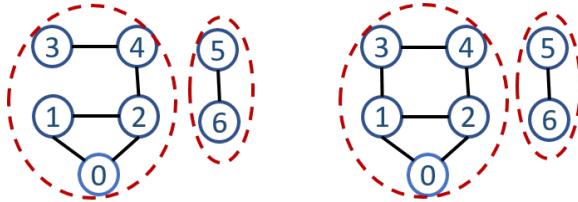


Figure 20.4: The connected components in undirected graph, each dashed red circle marks a connected component.

undirected graph in Fig. 20.4 has two connected components: $\{0, 1, 2, 3, 4\}$ and $\{5, 6\}$.

Given a directed graph,

- the term *Strongly Connected Component (SCC)* or *diconnected* is used to refer to the same definition– where in a SCC any two vertices are reachable to each other by paths. In the leftest directed graph shown in Fig. 20.5, there is a total of five SCCs: $\{0, 1, 2\}$, $\{3\}$, $\{4\}$, $\{5\}$, and $\{6\}$. Vertex 5 and 6 is only connected in one way, resulting into two separate SCCs.
- ignoring the direction of edges, a *weakly connected component (WCC)* equates to a connected component in the resulting undirected graph.

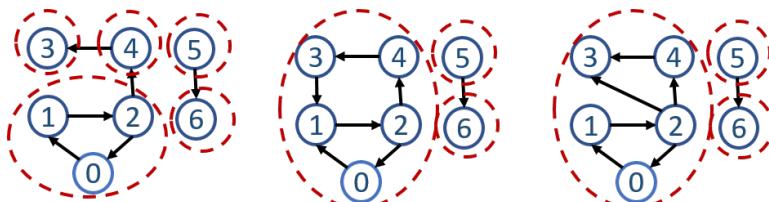


Figure 20.5: The strongly connected components in directed graph, each dashed red circle marks a strongly connected component.

Cycles and Strongly Connected Components A directed graph is acyclic if and only if it has no strongly connected subgraphs with more than one vertex. We call SCCs with at least two vertices nontrivial SCCs. Non-trivial SCCs contains at least one directed cycle, and more specifically, non-trivial SCCs is composed of a set of directed cycles as we have observed that there are two directed cycles in our above example and they share at least one common vertex. The shared common vertex act as “transferring stop” between these directed cycles thus they all compose to one component. Therefore, SCCs algorithms can be indirectly used to detect cycles. If there exists nontrivial SCC, directed graph contains cycles.

20.3.1 Connected Components Detection

In general, there are two ways to detect connected components in an undirected graph: graph search and union-find, each suits different needs.

Graph Search and Search Tree In undirected graph G , executing a BFS or DFS starting at some vertex u will result in a rooted search tree. As the edges are undirected or bidirectional, all vertices in the search tree belong to the same connected component. To find all connected components, we simply loop through all vertices V , for each vertex u :

- if u is not visited yet, we start a new DFS/BFS. Mark all vertices along the traversal as the same component.
- otherwise, u is already included in a previously found connected component, continue.

The time complexity will be $O(|V| + |E|)$ and the space complexity will be $O(|V|)$. Since the code is trivial, we only demonstrate it in the notebook.

Union Find

We represent each connected component as a set. For the exemplary graph in Fig. 20.4, we have two sets: $0, 1, 2, 3, 4$ and $5, 6$. Unlike the graph-search based approach, where the edges are visited in certain order,in union-find approach, the ordering of edges to be visited can be arbitrary. The algorithm using union-find is:

- Initialize in total $|V|$ sets, one for each vertex V .
- For each edge (u, v) in E , union the two sets where vertex u and v previously belongs to.

Implementing it with Python:

```

1 from collections import defaultdict
2 def connectedComponent(g):
3     n = len(g)
4     # initialize disjoint set
5     ds = DisjointSet(n)
6
7     for i in range(n):
8         for j in g[i]: # for edge i<->j
9             ds.union(i, j)
10    return ds.get_num_sets(), ds.get_all_sets()

```

How we implement the union-find data structure decides the complexity of this approach. For example, if we use linked list based structure, the complexity will be $O(|E| \times |V|)$ as we traversal $|V|$ edges and each step in worst case can take $O(|V|)$ to find the set that it belongs to. However, if path compression and union by rank is used for optimization, the time complexity could be lowered to $O(|E| \times \log |V|)$.

Dynamic Graph Since union-find has worse time complexity compared with graph search, then why do we care about it? The answer is: if we use graph search, whenever new edges and vertices are added to the graph, we have to rerun the graph search algorithm. Imagine that if we double $|V|$ and $|E|$, the worst time complexity will be $O(|V| \times (|V| + |E|))$, bringing up the complexity to polynomial of the number of edges. However, for each additional edge, union-find adds only a single merge operation to address the change, keeping the time complexity unchanged.

In detail, we adapt the union-find structure dynamically. Set up a `dict` to track vertex and its index in the union find. Set `index=0`. When a new edge (u, v) comes, union find includes:

- check if u and v exists in `dict`. If not, (a) add a key-value into the node tracker, (b) append `index` into the list of vertex-set, (c) `index+=1`.
- `find` the sets where u and v belongs to.

Implementation Here we demonstrate how to implement a dynamic connected component detection algorithm. First, convert the graph representation from adjacent list to a list of edges:

```

1 ug_edges = [(0, 1), (0, 2), (1, 2), (2, 4), (4, 3), (4, 3), (5,
6)]

```

Then, we implement a class `DynamicConnectedComponent` offering all functions needed.

```

1 class DynamicConnectedComponent():
2     def __init__(self):
3         self.ds = DisjointSet(0)
4         self.node_index= defaultdict(int)
5         self.index_node = defaultdict(int)

```

```

6     self.index = 0
7
8     def add_edge(self, u, v):
9         if u not in self.node_index:
10            self.node_index[u], self.index_node[self.index] = self.
11            index, u
12            self.ds.p.append(self.index)
13            self.ds.n += 1
14            self.index += 1
15
16            if v not in self.node_index:
17                self.node_index[v], self.index_node[self.index] = self.
18                index, v
19                self.ds.p.append(self.index)
20                self.ds.n += 1
21                self.index += 1
22                u, v = self.node_index[u], self.node_index[v]
23                self.ds.union(u, v)
24
25        return
26
27    def get_num_sets(self):
28        return self.ds.get_num_sets()
29
30    def get_all_sets(self):
31        sets = self.ds.get_all_sets()
32        return {self.index_node[key] : set([self.index_node[i] for i
33            in list(value)]) for key, value in sets.items()}


```

Now, to find the connected components dynamically based on incoming edges, we can run:

```

1 dcc = DynamicConnectedComponent()
2 for u, v in ug_edges:
3     dcc.add_edge(u, v)
4 dcc.get_num_sets(), dcc.get_all_sets()


```

The output is consistent with previous result, which is:

```

1 (2, {3: {0, 1, 2, 3, 4}, 6: {5, 6}})


```

Examples

1. 547. Number of Provinces(medium)
2. 128. Longest Consecutive Sequence (hard), union find solution: <https://leetcode.com/problems/longest-consecutive-sequence/discuss/1109808/Python-Clean-Union-Find-with-explanation>



Implement WCC detection algorithm in directed graph?

.

20.3.2 Strongly Connected Components

In graph theory, two nodes $u, v \in V$ are called strongly connected iff v is reachable from u and u is reachable from v . If we contract each SCC into a single vertex, the resulting graph will be a DAG. Denoting the contracted DAG as $G^{SCC} = (V^{SCC}, E^{SCC})$, V^{SCC} are vertices of SCCs and E^{SCC} are defined as follows:

(C_1, C_2) is an edge in G^{SCC} iff $\exists u \in C_1, v \in C_2$. (u, v) is an edge in G .

In other words, if there is an edge in G from any node in C_1 to any node in C_2 , there is an edge in G^{SCC} from C_1 to C_2 .

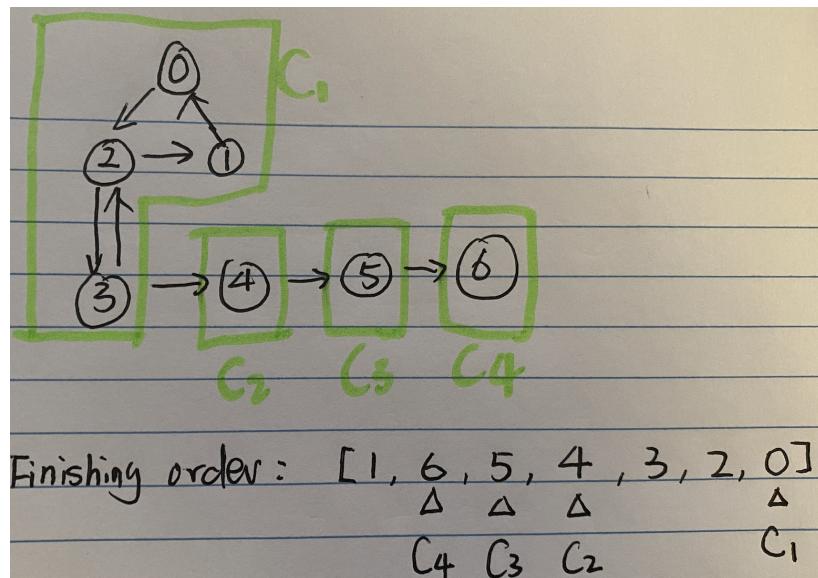


Figure 20.6: A graph with four SCCs.

Kosaraju's Algorithm If we were to do a DFS in G , and $C_1 \rightarrow C_2$ is an edge in G^{SCC} , then at least one vertex in C_1 will finish after all vertices in C_2 being finished. If we first start with vertex 0, the finishing order of all vertices is [1, 6, 5, 4, 3, 2, 0]. 0 finished later than 4 from C_2 , satisfying the claim. If we look purely at the last node from each SCC to turn dark, we get a topological sort of G^{SCC} in reverse([1, 6, 5, 4, 3, 2, 0]), which is [C4, C3, C2, C1]. How to find the last node in each SCC? We can reverse the dfs finishing order, having [0, 2, 3, 4, 5, 6, 1].

If we reverse the order, we have [0, 2, 3, 4, 5, 6, 1]. What happens if we do another round of DFS on the given ordering? First, starting from 0 (last node), we can (1) reach to all vertices in C_1 as they are connected, (2) reach to vertices in C_2 if there exists no edge or edges only from C_1 to C_2 in between. If we can reverse the edges in between, then we can avoid (2) and still keeps (1). The way we do this is: reverse the direction of all edges

in graph G . Run DFS on the reversed finishing ordering, then a SCC will include any vertex along the traversal that hasn't been put into a SCC yet. In our example, the process is:

```
0: find {0, 1, 2, 3}
4: find {4}
5: find {5}
6: find {6}
```

We formalize Kosaraju's algorithm into three steps:

1. Retrieve a reversed finishing order of vertices during DFS L . This step is similar to topological sort in an DAG.
2. Transpose the original graph G to G^T by reversing the directional of edges in G .
3. Run another DFS in L_1 ordering on G^T , any df-search tree starting from a vertex that hasn't been put into a SCC yet make up to another SCC.

Implementation The main function `scc` calls two functions: `topo_sort_scc` and `reverse_graph` to get L and G^T . The topological ordering like function:

```
1 # DFS traversal with reversed complete orders
2 def dfs(g, s, colors, complete_orders):
3     colors[s] = STATE.gray
4     for v in g[s]:
5         if colors[v] == STATE.white:
6             dfs(g, v, colors, complete_orders)
7     colors[s] = STATE.black
8     complete_orders.append(s)
9     return
10
11 # topologically sort in terms of the last node of each scc
12 def topo_sort_scc(g):
13     v = len(g)
14     complete_orders = []
15     colors = [STATE.white] * v
16     for i in range(v): # run dfs on all the node
17         if colors[i] == STATE.white:
18             dfs(g, i, colors, complete_orders)
19     return complete_orders[:: -1]
```

The main `scc` is straightforward:

```
1 # get conversed graph
2 def reverse_graph(g):
3     rg = [[] for i in range(len(g))]
4     for u in range(len(g)):
5         for v in g[u]:
6             rg[v].append(u)
7     return rg
```

```

8
9 def scc(g):
10    rg = reverse_graph(g)
11    orders = topo_sort_scc(g)
12
13    # track states
14    colors = [STATE.white] * len(g)
15    sccs = []
16
17    # traverse the reversed graph
18    for u in orders:
19        if colors[u] != STATE.white:
20            continue
21        scc = []
22        dfs(rg, u, colors, scc)
23        sccs.append(scc)
24    return sccs

```



Try to take a look at Tarjans' algorithm for SCC

Examples

1. 1520. Maximum Number of Non-Overlapping Substrings (hard): set up 26 nodes for all letters. A node represents a substray from start to end. Given a string abacdb, for a(0-2), add an edge between a -> to any other letter between start and end. Then we will have a directed graph. There is a scc (loop) between a and d, meaning a substring a has occurence of b and b substring has occurence of a, which is conflicting condition 2, so that they have to be combined. all results are sccs that are leaves in the contracted scc graph. We can think the scc graph is acyclic which is a forest. If we choose an internal node, we cant choose any of the leaves. Which making choosing the number of leaves maximum. Another solution is using two pointers: <https://zxi.mytechroad.com/blog/greedy/leetcode-1520-maximum-number-of-non-overlapping>

20.4 Minimum Spanning Trees

Problem Definition A *spanning tree* in an undirected graph $G = (V, E)$ is a set of edges, with no cycles, that connects all vertices. There can exist many spanning trees in a graph. Given a weighted graph, we are particularly interested with the *minimum spanning tree (MST)*—a spanning tree with the least total edge cost.

One example is shown in Fig. 20.7. This graph can represent a collection of houses, and possible wires that we can lay. How we lay wires to connect all houses with the least total cost is equivalently a MST problem.

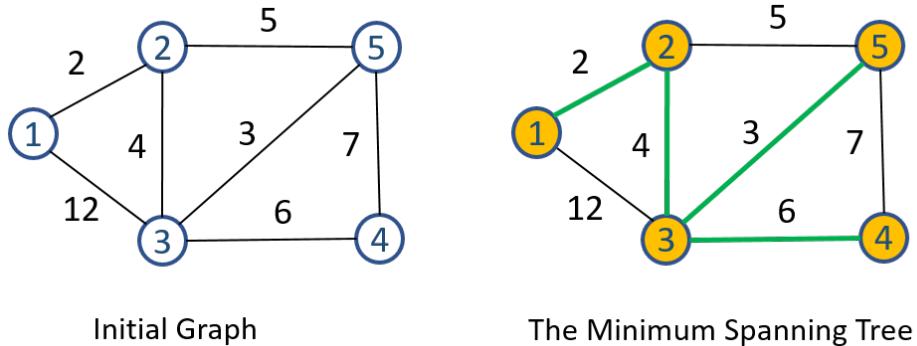


Figure 20.7: Example of minimum spanning tree in undirected graph, the green edges are edges of the tree, and the yellow filled vertices are vertices of MST (change this to a graph with multiple spanning tree, and highlight the one with the minimum ones).

Spanning Tree To obtain a tree from a graph, the essence is to select edges iteratively until we have $|V| - 1$ edges which form a tree connecting V . We have two general approaches:

- Start with a forest consists of $|V|$ trees and contains only one node. We design a method to merge these trees into a final connected MST by selecting one edge at a time. This is the path taken by the Kruskal's algorithm.
- Start with a root node which can be any vertex selected from G , grow the tree by spanning to more nodes iteratively. In the process, we maintain two disjoint sets of vertices: one containing vertices that are in the growing spanning tree S and the other to track all remaining vertices $V - S$. This is the path taken by the Prim's algorithm.

We denote the edges in the growing as A . In this section, we explain two greedy algorithms to find MST.

20.4.1 Kruskal's Algorithm

Kruskal's algorithm starts with $|V|$ trees that each has only one node. The main process of the algorithm is to merge these trees into a single one by iterating through all edges.

Generate Spanning Tree with Union-Find For each edge (u, v) :

- if u and v belongs to the same tree, adding this edge will form a cycle, thus we discard this edge.

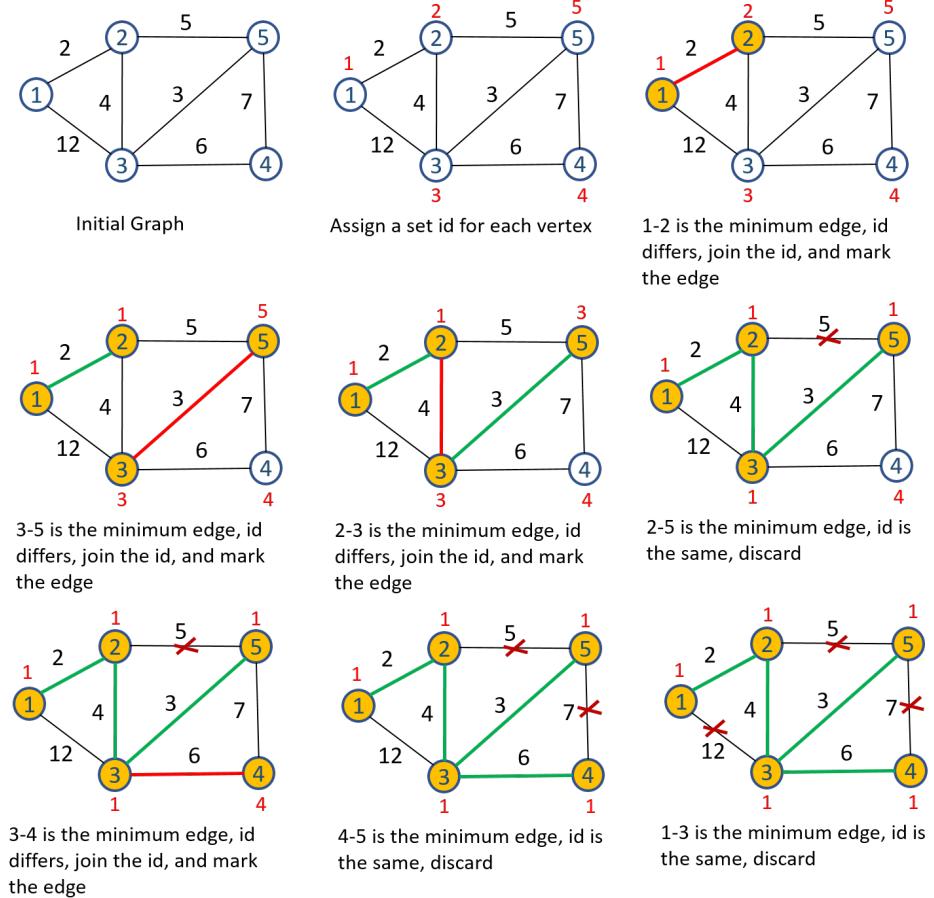


Figure 20.8: The process of Kruskal's Algorithm

- otherwise, combine these two trees and add this edge into A .

This process will result in a single spanning tree. In implementation wise, we can do this easily by using union-find data structure. A tree is a set. Adding one edge is to merge two sets/trees into a single one if they belong to different sets.

Being Greedy with MST At each step i , we have $|E| - i$ edges to choose from. Applying the principle of greedy algorithm, maybe we can try to choose the edge with the minimum cost among $|E| - i$ options. That is to say, we iterate edges in increasing order of its weight in the process of generating a spanning tree. Doing so will ensure us to have the MST, and this algorithm is the so called Kruskal's algorithm.

Fig. 20.8 demonstrates the run of Kruskal's on the input undirected graph. Here, the edges are ordered increasingly, i.e., $[(1,2), (3, 5), (2, 3),$

$(2, 5), (3, 4), (4, 5), (1, 3)$. As initialization, we assign a set id for each vertex that is marked in red and placed above its corresponding vertex. The process is:

edge	logic	action
$(1, 2)$	$1's \text{ set_id } 1 \neq 2's \text{ set_id } 2$	merge set 2 to set 1
$(3, 5)$	$3's \text{ set_id } 3 \neq 5's \text{ set_id } 5$	merge set 5 to set 3
$(2, 3)$	$2's \text{ set_id } 1 \neq 3's \text{ set_id } 3$	merge set 3 to set 1
$(2, 5)$	$2's \text{ set_id } 1 == 5's \text{ set_id } 1$	continue
$(3, 4)$	$3's \text{ set_id } 1 \neq 4's \text{ set_id } 4$	merge set 4 to set 1
$(4, 5)$	$4's \text{ set_id } 1 == 5's \text{ set_id } 1$	continue
$(1, 3)$	$1's \text{ set_id } 1 == 3's \text{ set_id } 1$	continue

This process produces edges $[(1, 2), (3, 5), (2, 3), (3, 4)]$ as the edges of the final MST. We can have slightly better performance if we can stop iterating through edges once we have selected $|V| - 1$ edges. The implementation is as simply as:

```

1 from typing import Dict
2 def kruskal(g: Dict):
3     # g is a dict with node: adjacent nodes
4     vertices = [i for i in range(1, 1 + len(g))]
5     vertices = g.keys()
6     n = len(vertices)
7     ver_idx = {v: i for i, v in enumerate(vertices)}
8
9     # initialize a disjoint set
10    ds = DisjointSet(n)
11
12    # sort all edges
13    edges = []
14    for u in vertices:
15        for v, w in g[u]:
16            if (v, u, w) not in edges:
17                edges.append((u, v, w))
18    edges.sort(key=lambda x: x[2])
19
20    # main section
21    A = []
22    for u, v, w in edges:
23        if ds.find(ver_idx[u]) != ds.find(ver_idx[v]):
24            ds.union(ver_idx[u], ver_idx[v])
25            print(f'{u} → {v}: {w}')
26            A.append((u, v, w))
27    return A

```

For the exemplary graph, we denote an weighted edge as a (key, value) pair, where the value is a tuple of two with the first item being the other endpoint from the key vertex and the second item being the weight of the edge. The graph will thus be represented by a dictionary, $\{1:[(2, 2), (3, 12)], 2:[(1, 2), (3, 4), (5, 5)], 3:[(1, 12), (2, 4), (4, 6), (5, 3)], 4:[(3, 6), (5, 7)], 5:[(2, 5), (3, 3), (4, 7)]\}$. Running `kruskal(a)` will return the following edges:

```
[(1, 2, 2), (3, 5, 3), (2, 3, 4), (3, 4, 6)]
```

Complexity Analysis The sorting takes $O(|E| \log |E|)$ big oh time. The cost of checking each edge's belonging set id and merging two trees into a single one is decided by the complexity of the disjoint set, it can range from $O(\log |V|)$ to $O(|V|)$. Therefore, we can conclude the time complexity will be bounded by the sorting time, i.e., $O(|E| \log |E|)$.

20.4.2 Prim's Algorithm

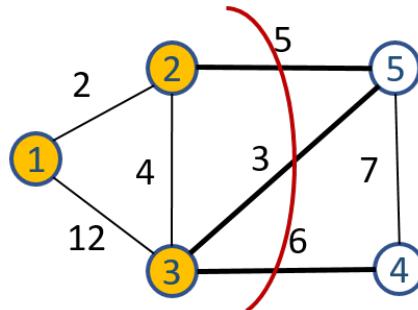


Figure 20.9: A cut denoted with red curve partition V into $\{1,2,3\}$ and $\{4,5\}$.

In graph theory, a *cut* is a partition of V into S and $V - S$. For example, in Fig. 20.9 a cut is marked by red curve, removing three edges $(2, 5), (3, 5), (3, 4)$ partitions the set into two subgraph with subsets $\{1, 2, 3\}$ and $\{4, 5\}$. A *cross edge* $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its endpoint is in S and the other is in $V - S$. A *light edge* is the minimum edge among all cross edges, such as edge $(3, 5)$ is the light edge in our example. We say, a cut *respects* a set of edges A if no edge in A crosses the cut, such as the marked cut in the example respects the set of edges $(1, 2), (2, 3), (1, 3)$.

Prim's algorithm starts with a randomly chosen root node and be put into a set S , leaving us with two sets of vertices, S and $V - S$. Next, it iteratively grows the partial and connected MST by adding an edge from the cross edges between the cut of $(S, V - S)$. Prim's algorithm is greedy in the sense that it chooses a light edge among its options to form the final MST. This process simulates the uniform-cost search which compose the Dijkstra's shortest path algorithm.

Fig. 20.10 demonstrates the process of Prim's algorithm. We start from vertex 1. with the set $A, S, V - S$, the cross edges at each step are denoted as CE , and a decision valid if it does not form a cycle within A , we list the process as:

A	S	$V - S$	CE	light edge
	1	2, 3, 4, 5	(1, 2), (1, 3)	(1, 2)
(1, 2)	1, 2	3, 4, 5	(1, 3), (2, 3), (2, 5)	(2, 3)
(1, 2), (2, 3)	1, 2, 3	4, 5	(3, 4), (3, 5), (2, 5)	(3, 5)

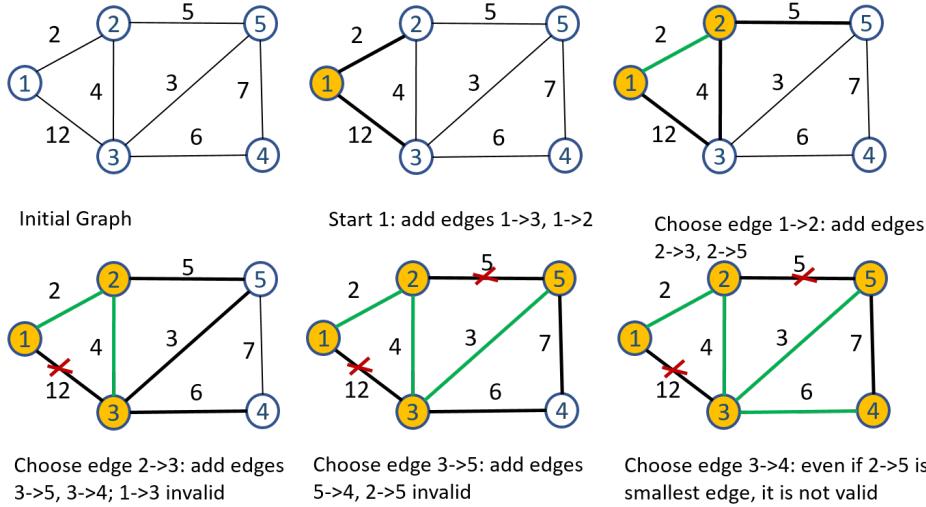


Figure 20.10: Prim's Algorithm, at each step, we manage the cross edges.

$(1, 2), (2, 3), (3, 5)$	$1, 2, 3, 5$	4	$(3, 4), (5, 4)$	$(3, 4)$
$(1, 2), (2, 3), (3, 5), (3, 4)$	$1, 2, 3, 4, 5$			

Implementation

One key step is to track all valid cross edges and be able to select the minimum edge from the set. Naturally, we use priority queue pq . pq can be implemented in two ways:

- **Priority Queue by Edges**—Considering the set S as a frontier set, pq maintains all edges expanded from the frontier set.
- **Priority Queue by Vertices**— pq maintains the minimum cross edge cost between vertices in S to the current vertex which is in $|V - S|$. This is an optimization over the first approach as it reduces multiple cross edges between S and current vertex v into a single cost – the minimum.

Priority Queue by Edges For example shown in Fig. 20.10, at first, the frontier set has only 1, then we have edges $(1, 2), (1, 3)$ in pq . Once edge $(1, 2)$ is popped out as it has the smallest weight, we explore all outgoing edges of vertex 2 to nodes in $V - S$, adding $(2, 3), (2, 5)$ in pq , resulting $\text{pq} = (2, 3), (2, 5), (1, 3)$. Then we pop out edge $(2, 3)$, and explore outgoing edges of vertex 3 and add $(3, 4), (3, 5)$ into pq , with $\text{pq} = (2, 5), (1, 3), (3, 4), (3, 5)$. At this moment, we can see that edge $(1, 3)$ is no longer a cross edge. Therefore, whenever we are about to add the light edge into the expanding tree, we check if both of its endpoints are in set S already. If true, we skip this

edge and use the next valid light edge. Repeat this process will get us the set of edges A forming a MST. The Python code is as:

```

1 import queue
2
3 def _get_light_edge(pq, S):
4     while pq:
5         # Pick the light edge
6         w, u, v = pq.get()
7         # Filter out non-cross edge
8         if v not in S:
9             S.add(v)
10            return (u, v, w)
11        return None
12
13 def prim(g):
14     cur = 1
15     n = len(g.items())
16     S = {cur} #spanning tree set
17     pq = queue.PriorityQueue()
18     A = []
19
20     while len(S) < n:
21         # Expand edges for the exploring vertex
22         for v, w in g[cur]:
23             if v not in S:
24                 pq.put((w, cur, v))
25
26         le = _get_light_edge(pq, S)
27         if le:
28             A.append(le)
29             cur = le[1] #set the exploring vertex
30         else:
31             print(f'Graph {g} is not connected.')
32             break
33     return A

```

In line 24, we use a 3 item tuple representing the edge cost, the first endpoint in the set S and the second endpoint in $V - S$ to align with the fact that the `PriorityQueue()` uses the first item of a tuple as the key for sorting. The `while` loop is similar to our breath-first-search and can be terminated in the following two conditions:

- when the set S is as large as the set V by checking the size of set S
- when we can not find a light edge which happens when the graph is not connected.

Call `prim(a)` will return us the following A :

```
[(1, 2, 2), (2, 3, 4), (3, 5, 3), (3, 4, 6)]
```

Complexity Analysis The main cost of this implementation is on the priority queue, which has a maximum of $|E|$ items. In the worst case we have to enqueue and dequeue all edges, making the complexity as $O(|E|\log |E|)$. In a graph, generally, $E < V^2$, the complexity become $O(|E|\log V)$.

Priority Queue by Vertices Instead of tracking cross edges in the priority queue explicitly, we reduce all cross edges that reaches to a vertex in $V - S$ into the smallest cost and a predecessor which is to track the node in S that resulted in the smallest cost, saving us some additional space and time in the queue operations.

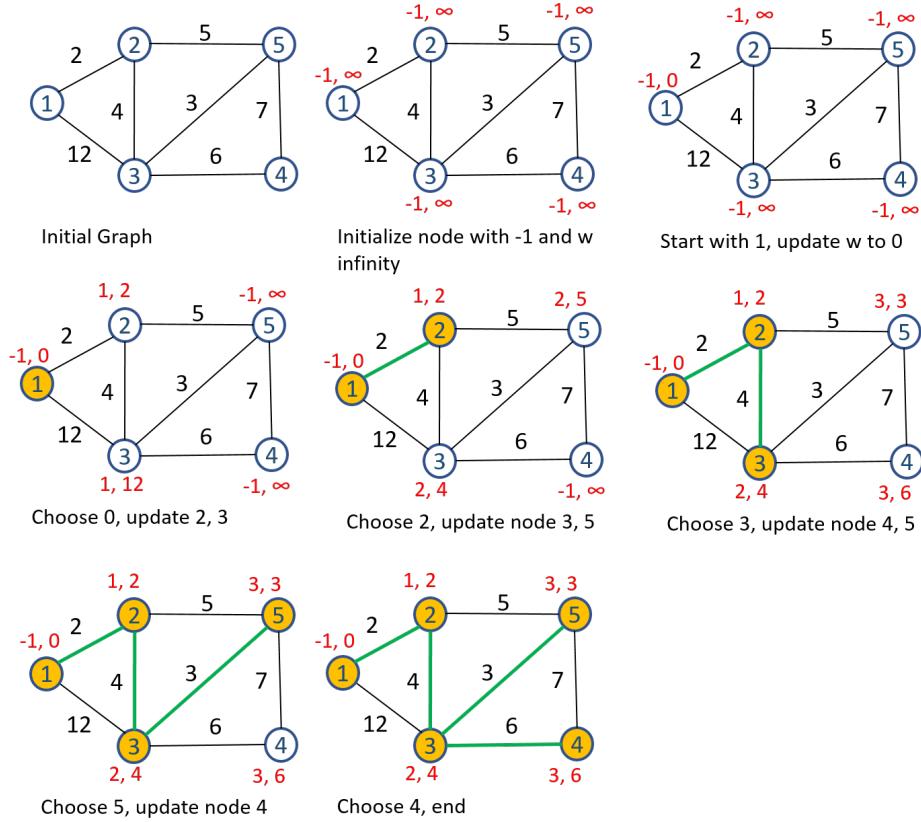


Figure 20.11: Prim's Algorithm

As shown in Fig. 20.11, we first initialize a priority queue with $|V|$ items, each has a task id same as the vertex id, a predecessor vertex $p = -1$, and a cost initialized with ∞ . We start by pointing vertex 1 as the root node, setting $S = 1$ and modify the task 1's cost to 0 and points its predecessor to itself. Then, we repeatedly pop out the vertex in the queue that has the smallest weight, along with the predecessor of this node, we are choosing the light edge. With this chosen node, we are able to reach out to adjacent

nodes that are still in $V - S$ and see if we are able to find an even “lighter” edge. Applying this process on the given example:

1. First, we have the start vertex 1 with the smallest cost, pop it out, and explore edges $(1, 2), (1, 3)$, resulting in (a) modifying task 2 and 3’s cost to 2 and 12, respectively and (b) set 2 and 3’s predecessor to 1.
2. Pop out vertex 2, explore edges $(2, 3), (2, 5)$, resulting in (a) modifying task 3 and 5’s cost to 4 and 5, respectively and (b) set 3 and 5’s predecessor to 2.
3. Pop out vertex 3, explore edges $(3, 5), (3, 4)$, resulting in (a) modifying task 5 and 4’s cost to 3 and 6, respectively and (b) set 3 and 5’s predecessor to 3.
4. Pop out vertex 5, explore edges $(5, 4)$: since the new cross edge $(5, 4)$ has larger cost compared with previous reduced cross edge to reach to vertex 4, the vertex 4 in the queue is not modified.
5. Pop out vertex 4, no more new edges to expand, terminate the program.

This process results in the exactly same MST compared with the implementation by edges. However, it adds additional challenges into the implementation of the priority queue: We have to modify an enqueued item’s record during the life cycle of the queue. In the Python implementation, we use the our customized `PriorityQueue()` in Section. ??(also included in the notebook). The main process of the algorithm is:

```

1 def prim2(g):
2     n = len(g.items())
3     pq = PriorityQueue()
4     S = {}
5     A = []
6     # Initialization
7     for i in range(n):
8         pq.add_task(task=i+1, priority=float('inf'), info=None) # task: vertex, priority: edge cost, info: predecessor vertex
9
10    S = {1}
11    pq.add_task(1, 0, info=1)
12
13    while len(S) < n:
14        u, p, w = pq.pop_task()
15        if w == float('inf'):
16            print(f'Graph {g} is not connected.')
17            break
18        A.append((p, u, w))
19        S.add(u)

```

```

20   for v, w in g[u]:
21     if v not in S and w < pq.entry_finder[v][0]:
22       pq.add_task(v, w, u)
23
24   return A
25

```

Calling function `prim2(a)` will output the following A :

```

1  [(1, 1, 0), (1, 2, 2), (2, 3, 4), (3, 5, 3), (3, 4, 6)]
2

```

Examples

1. 1584. Min Cost to Connect All Points (medium)
2. 1579. Remove Max Number of Edges to Keep Graph Fully Traversable (hard)



Try to prove the correctness of Kruskal's Algorithm.

20.5 Shortest-Paths Algorithms

Problem Definition Given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow R$ that maps edges to real-valued weights, the weight of a path $p = (v_0, v_1, \dots, v_k)$ is the summation over its constituent edge weights, denoted as $w(p)$:

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i) \quad (20.1)$$

The *shortest path* problem between v_i and v_j is to find the shortest path weight $\sigma(v_i, v_j)$ along with the shortest path p .

$$\sigma(v_i, v_j) = \begin{cases} \min\{w(p) : v_i \xrightarrow{p} v_j\} & \text{if there is a path from } v_i \text{ to } v_j \\ \infty & \text{otherwise} \end{cases} \quad (20.2)$$

For example, for the graph shown in Fig. 20.12, the shortest-path weight and its corresponding shortest-path between s to any other vertex in V is listed as:

(source, target)	shortest-path weight	shortest path
(s, s)	0	s
(s, y)	7	(s, y)
(s, x)	4	(s, y, x)
(s, t)	2	(s, y, x, t)
(s, z)	-2	(s, y, x, t, z)

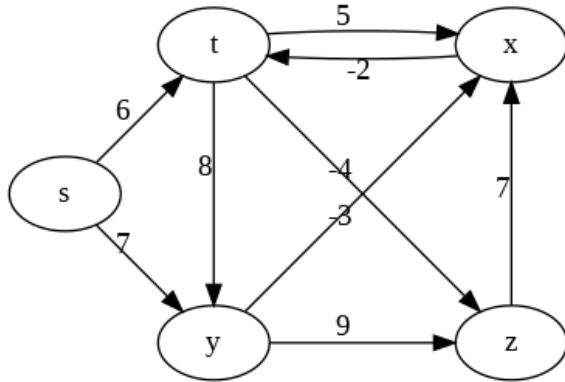


Figure 20.12: A weighted and directed graph.

Variants of Shortest-path Problems Generally, there exists a few variants of shortest path problems:

1. *Single-source shortest-path*: Find a shortest path from a given source s vertex to each vertex $v \in V$.
2. *Single-target shortest-path*: Find a shortest path to a given target t from each vertex $v \in V$. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source shortest-path problem.
3. *Single-pair shortest-path problem*: Find a shortest path from u to v for given vertices u and v . If we solve the single-source problem with source vertex u , we solve this problem too.
4. *All-pairs shortest-path problem*: Find a shortest path from u to v for every pair of vertices u and v in V if there exists one. Although we can solve this problem by running a single-source algorithm once for each vertex, we usually can solve it faster with algorithms addressed in Section (Sec. 20.5.4).

20.5.1 Algorithm Design

In this section, we discuss the shortest path problem, and analyze it by using both graph theory and the fundamental algorithm design principle—Dynamic Programming.

Shortest path and Cycle From our experience in Combinatorial Search, we have to detect cycles within a path in the graph-based tree search to avoid being stuck in infinite recursion. So, how will cycle affect the detection of shortest paths? For example, in Fig. 20.12, a path $p = (s, t, x, t)$ contains

the cycle (t, x, t) . Because the cycle has a positive path weight $5 + (-2)$, the path (s, t) remains smaller than the path that comes with the cycle. However, if we switch the weight of edge (t, x) with that of (x, t) , then the same cycle (t, x, t) will have negative path weight $(-5) + 2$, repeating the cycle within the path infinitely we will have a cost of $-\infty$. Therefore, for a graph where the weights can be both negative and positive, one requirement posed on the single-source shortest-path algorithm, recursive or iterative, is to detect the negative-weight cycle that is reachable from the source. Once we get rid of all negative-weight cycles, the remaining of the algorithm can focus on only shortest-paths of at most $|V| - 1$ edges, and the resulting shortest-paths will not contain neither negative- nor positive-weight cycles.

Exponential Naive Solution

Assume the given graph has no negative-weight cycle, a naive solution to obtain the shortest path and its weight is simply through a tree-search which starts from a source vertex s and enumerates all possible paths between s to any other vertex in V . The search tree will have a maximum height of $|V| - 1$, making the time complexity of this naive solution to be $O(b^{|V|})$, where b is the maximum branch of a vertex. Recall the path enumeration in Search Strategies, we implement this solution as:

```

1 def all_paths(g, s, path, cost, ans):
2     ans.append({'path': path[:], 'cost': cost})
3     for v, w in g[s]:
4         # Avoid cycle
5         if v in path:
6             continue
7         path.append(v)
8         cost += w
9         all_paths(g, v, path, cost, ans)
10        cost -= w
11        path.pop()

```

To obtain all possible paths, we call the function `all_paths()` with the following code:

```

1 g = {
2     't':[( 'x', 5), ( 'y', 8), ( 'z', -4)],
3     'x':[( 't',-2)],
4     'y':[( 'x',-3), ( 'z',9)],
5     'z':[( 'x',7)],
6     's':[( 't', 6), ( 'y', 7)],
7 }
8 ans = []
9 all_paths(g, 's', [ 's'], 0, ans)

```

Shortest-paths Tree We visualize all paths in `ans` in a tree structure shown in Fig. 20.13. We can easily extract the shortest paths between s

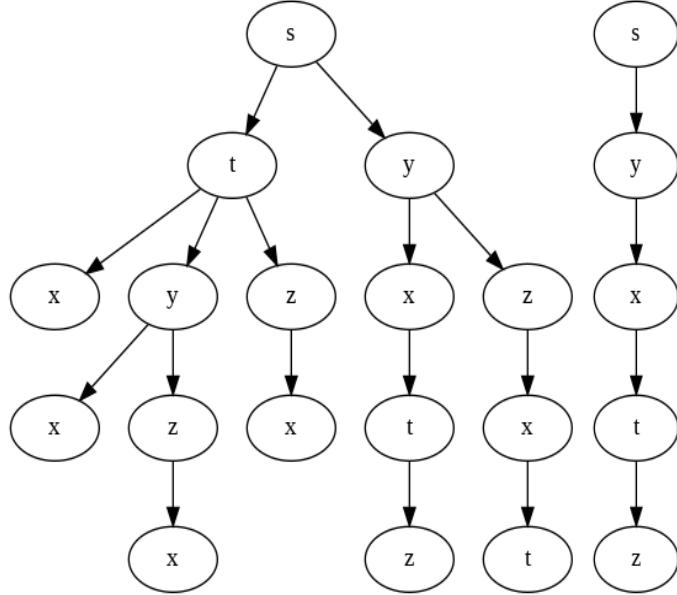


Figure 20.13: All paths from source vertex s for graph in Fig. 20.12 and its shortest paths.

to any other vertex from this result, which is shown on the right side of Fig. 20.13. All possible paths starting from source vertex can be viewed as a tree, and the shortest paths from source to all other vertices within the graph will be a subtree of the former tree structure, known as the *shortest-paths tree*. Formally, a shortest-paths tree rooted at s is a directed subgraph $G' = (V', E')$, where $V' \in V$ and $E' \in E$, such that

1. V' is the set of vertices reachable from s in G ,
2. for each $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G .

Predecessor Rule The shortest-paths tree makes it possible for us to track shortest paths with the predecessor rule: Given a graph $G = (V, E)$, and in the single-source shortest path problem, we maintain for each vertex $v \in V$ a predecessor π that is either another vertex or empty as for the root node. The shortest-paths between s and another vertex v can be obtained by iterating the chained predecessors starting from v and all the way backward to the source s . To summarize, each vertex v in the graph stores two values, $d(v)$ and $\pi(v)$, which (inductively) describe a tentative shortest path from s to v .

Optimization

As we see, shortest path problem is a truly combinatorial optimization problem, making them the best demonstration examples of the algorithm design principles—Dynamic Programming and Greedy Algorithm. On the other hand, depending on the characteristics of targeting graph, either they are dense or spares, directed acyclic graph (DAG) or not DAG, we can further optimize the efficiency besides of the design principle. However, in this chapter, we focus on the gist: *how to solve all-pair shortest path problems with dynamic programming?*

First, we use an adjacency matrix to represent our weight matrix W of size $|V| \times |V|$. In the process, we track shortest-path weight estimate D and additionally the predecessor Π . Both D and Π are of same size as W . w_{ij} indicates the weight of each edge with startpoint i and endpoint j ,

$$W(i, j) = \begin{cases} 0 & \text{if } i = j \\ w_{ij} & \text{if } i \neq j, \text{ and } (i, j) \in E \\ \infty & \text{if } i \neq j, \text{ and } (i, j) \notin E \end{cases} \quad (20.3)$$

With this definition, we show a naive directed graph in Fig. 20.14 along with its W .

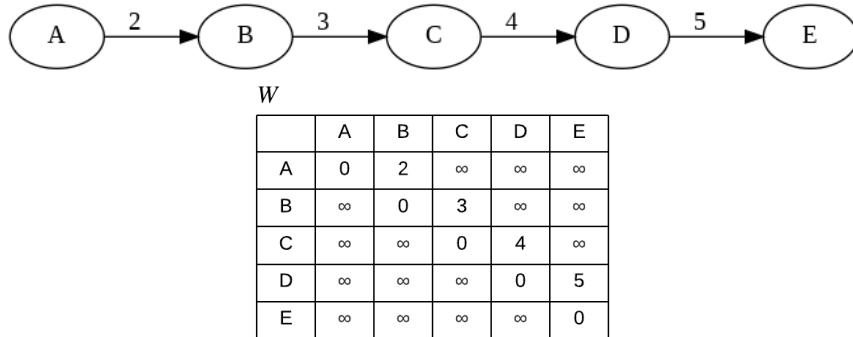


Figure 20.14: The simple graph and its adjacency matrix representation (changing it to lower letter)

Overlapping Subproblems and Optimal Substructures For all-pair shortest paths, we have $|V|^2$ optimal subproblems, each subproblem $D(i, j)$ is defined as the shortest-path between v_i and v_j . Optimal Substructures states “*the optimal solution to a problem has the optimal solutions to subproblems in it.*” All of this boils down to how to define the “subproblem” and how a larger subproblem is divided into smaller subproblems (the recurrence relation).

With our naive directed graph, the shortest path between a and d come from the shortest path between a to an intermediate node x or the shortest

path between a and d found so far. First, we define the subproblem as the shortest path between a and d with maximum path length(MPL) m . With this definition, we show two possible ways of dividing the subproblem:

1. We divide a subproblem with MPL m into a subproblem with MPL $m - 1$ and an edge. Therefore, the shortest path at this maximum length m is either the shortest path found so far or equals to the shortest path between a and x plus the weight of edge (x, d) , our recurrence relation is:

$$D^m(a, d) = \min_x(D^{m-1}(a, d), D^{m-1}(a, x) + W(x, d)) \quad (20.4)$$

As we can see, each update for an item in distance matrix D takes $O(|V|)$ time as it has to check all possible intermediate nodes. Furthermore, it takes $|V| - 1$ passes to update D^0 all the way to $D^{|V|-1}$. Therefore, this approach has a time complexity of $O(|V|^4)$. We demonstrate the update process in Fig. 20.15 for our naive example.

(1).png (1).png.mps (1).png.pdf (1).png.png (1).png.jpg (1).png.mps
 (1).png.jpeg (1).png.jbig2 (1).png.jb2 (1).png.PDF (1).png.PNG
 (1).png.JPG (1).png.JPEG (1).png.JBIG2 (1).png.JB2 (1).png.eps

	D^1						D^2					
	A	B	C	D	E		A	B	C	D	E	
A	0	2	∞	∞	∞		A	0	2	5	∞	
B	∞	0	3	∞	∞		B	∞	0	3	7	∞
C	∞	∞	0	4	∞		C	∞	∞	0	4	9
D	∞	∞	∞	0	5		D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0		E	∞	∞	∞	∞	0

	D^3						D^2					
	A	B	C	D	E		A	B	C	D	E	
A	0	2	5	9	∞		A	0	2	5	9	14
B	∞	0	3	7	12		B	∞	0	3	7	12
C	∞	∞	0	4	9		C	∞	∞	0	4	9
D	∞	∞	∞	0	5		D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0		E	∞	∞	∞	∞	0

Figure 20.15: DP process using Eq. 20.4 for Fig. 20.14

2. We divide a subproblem with MPL m into two equal sized subproblem, each with MPL $m/2$. Therefore, the shortest path at this maximum length m is either the shortest path found so far or equals to the shortest path between a and x of length $m/2$ plus the shortest path

(2).png (2).png.mps (2).png.pdf (2).png.png (2).png.jpg (2).png.mps
 (2).png.jpeg (2).png.jbig2 (2).png.jb2 (2).png.PDF (2).png.PNG
 (2).png.JPG (2).png.JPEG (2).png.JBIG2 (2).png.JB2 (2).png.eps

 D^1

	A	B	C	D	E
A	0	2	∞	∞	∞
B	∞	0	3	∞	∞
C	∞	∞	0	4	∞
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

 D^2

	A	B	C	D	E
A	0	2	5	∞	∞
B	∞	0	3	7	∞
C	∞	∞	0	4	9
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

 D^4

	A	B	C	D	E
A	0	2	5	9	14
B	∞	0	3	7	12
C	∞	∞	0	4	9
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

Figure 20.16: DP process using Eq. 20.5 for Fig. 20.14

between x and d of length $m/2$. With recurrence relation:

$$D^m(a, d) = \min_x(D^{m/2}(a, d), D^{m/2}(a, x) + D^{m/2}(x, d)) \quad (20.5)$$

Similarly, each update takes $|V|$ time. Differently, it only takes $\log |V|$ updates to get the final optimal subproblems. Thus, this approach gives a better time complexity, $O(|V|^3 \log |V|)$. The process is demonstrated in Fig. 20.16.

Alternatively, we define the subproblem as the shortest path between a and d with x as an intermediate node along the path, the number of intermediate node is $|V|$. Here, we use k to index the intermediate node, and i, j to index the start and end node. Then a subproblem $D^k(i, j)$ can be either the shortest path between i and j with intermediate nodes $0, 1, \dots, k - 1$ or the shortest path between i and k with all previous intermediate nodes plus the shortest path between k and j with all previous intermediate nodes. The recurrence relation is:

$$D^k(i, j) = \min(D^{\{0, \dots, k-1\}}(i, j), D^{\{0, \dots, k-1\}}(i, k) + D^{\{0, \dots, k-1\}}(k, j)) \quad (20.6)$$

As we see, each recurrence update only takes constant time. At the end, after we consider all possible intermediate nodes, we reach out to the optimal

solution. This approach results in the best time complexity, $O(|V|^3)$ so far. We demonstrate the update process in Fig. 20.17. At pass C , using C as intermediate node, we end up only use C -th row and C -th column to update our matrix.

(3).png (3).png.mps (3).png.pdf (3).png.png (3).png.jpg (3).png.mps
 (3).png.jpeg (3).png.jbig2 (3).png.jb2 (3).png.PDF (3).png.PNG
 (3).png.JPG (3).png.JPEG (3).png.JBIG2 (3).png.JB2 (3).png.eps

		D^A							D^B						
		A	B	C	D	E			A	B	C	D	E		
		A	0	2	∞	∞	∞			A	0	2	5	∞	∞
		B	∞	0	3	∞	∞			B	∞	0	3	∞	∞
		C	∞	∞	0	4	∞			C	∞	∞	0	4	∞
		D	∞	∞	∞	0	5			D	∞	∞	∞	0	5
		E	∞	∞	∞	∞	0			E	∞	∞	∞	∞	0

		D^C							D^D						
		A	B	C	D	E			A	B	C	D	E		
		A	0	2	5	9	∞			A	0	2	5	9	14
		B	∞	0	3	7	∞			B	∞	0	3	7	12
		C	∞	∞	0	4	∞			C	∞	∞	0	4	9
		D	∞	∞	∞	0	5			D	∞	∞	∞	0	5
		E	∞	∞	∞	∞	0			E	∞	∞	∞	∞	0

		D^E							
		A	B	C	D	E			
		A	0	2	5	9	14		
		B	∞	0	3	7	12		
		C	∞	∞	0	4	9		
		D	∞	∞	∞	0	5		
		E	∞	∞	∞	∞	0		

Figure 20.17: DP process using Eq. 20.6 for Fig. 20.14

As we shall see later, the first way is similar to Bellman-Ford, the second is a repeated squaring version of Bellman-Ford, and the third is Floyd-warshall algorithm.

Greedy algorithms For $|V|^2$ subproblems, solving each subproblem takes at least $|V|$ using Floyd-warshall algorithm. Greedy approach would think of ways to decide the optional solution to each subproblem in one try, making it $|V|^2$ or $|V| + |E|$. We will see Dijkstra algorithm which is only applicable on all positive weighted W .

In the following section, we start with going through algorithms solving

single-source shortest path problem before we put up more details to the all-pair shortest path algorithms introduced above.

20.5.2 The Bellman-Ford Algorithm

Bellman-ford algorithm addresses single-source shortest path problem using a single-source version of DP approach one.

Dynamic Programming Representation Given a single source node s in graph G , we define D and Π as just a one-dimensional vector instead of a matrix in all-pair shortest paths. D_i^m represents the shortest path between s and i with maximum path length m . When $m = 0$, there is a shortest path from s to v with no edge iff $s = v$.

$$D_i^0 = \begin{cases} 0 & \text{if } s = i \\ \infty & \text{otherwise} \end{cases} \quad (20.7)$$

Similarly, Π^0 is initialized as `None`. Our simplified recurrence relation is:

$$D_i^m = \min(D_i^{m-1}, \min_{k,k \in [0,n-1]}(D_k^{m-1} + W(k, i))) \quad (20.8)$$

which can be further simplified to:

$$D_i^m = \min_{k,k \in [0,n-1]}(D_k^{m-1} + W(k, i)) \quad (20.9)$$

In Eq. 20.9, once an intermediate node is found to have smaller tentative path weight than the current's value, we set $\Pi(i) = k$.

Implementation In function `bellman_ford_dp`, W is an $n \times n$ adjacency matrix. In the first `for` loop, we run recurrence relation in Eq. 20.9 for $|V| - 1$ passes, giving the fact that other than the negative-weight cycle, there will be at most $|V| - 1$ edges for all paths within the graph.

```

1 def bellman_ford_dp(s, W):
2     n = len(W)
3     # D, pi
4     D = [float('inf') if i != s else 0 for i in range(n)] # * n
5     P = [None] * n
6     for m in range(n-1):
7         newD = D[:]
8         for i in range(n): # endpoint
9             for k in range(n): # intermediate node
10                if D[k] + W[k][i] < newD[i]:
11                    P[i] = k
12                    newD[i] = D[k] + W[k][i]
13
14     D = newD
15     print(f'D{m+1}: {D}')
16     return D, P

```

Now, to retrieve the path from source s to other vertices, we implement a recursive function named `get_path` that starts from the target u and backtraces to the source s through Π . The code is as:

```

1 def get_path(P, s, u, path):
2     path.append(u)
3     if u == s:
4         print('Reached to the source vertex, stop!')
5         return path[::-1]
6     elif u is None:
7         print(f"No path found between {s} and {u}.")
8         return []
9     else:
10        return get_path(P, s, P[u], path)

```

For the graph in Fig. 20.12, the updating on D using s as source is visualized in Fig. 20.18. Connecting all red arrows along with the shaded gray nodes,

(2).png (2).png.mps (2).png.pdf (2).png.png (2).png.jpg (2).png.mps
(2).png.jpeg (2).png.jbig2 (2).png.jb2 (2).png.PDF (2).png.PNG
(2).png.JPG (2).png.JPEG (2).png.JBIG2 (2).png.JB2 (2).png.eps

	0	1	2	3	4
	t	x	y	z	s
D^0	∞	∞	∞	∞	0
D^1	6	∞	∞	7	0
D^2	6	4	7	2	0
D^3	2	4	7	2	0
D^4	2	4	7	-2	0

Figure 20.18: The update on D for Fig. 20.12. The gray filled spot marks the nodes that updated its estimate value, with its predecessor indicated by incoming red arrow.

we have a tree structure, each update on D , we expand the tree by one more level, updating the best estimate reaching to target node with one more possible edge. We visualize this tree structure in Fig. 20.21. We explain the tree like this: if we are at most one edge away from s , we get t as small as 6, if we are three edges away, t is able to gain a smaller value through its predecessor x which is at most 2 edges away. After the last round of update, when the tree reaches to height $|V| - 1$, the predecessor vector Π will give out the shortest-path tree: each edge in the shortest path tree can be obtained by connecting each predecessor with vertices in the graph. The shortest-path tree is marked in Fig. 20.21 in red color.

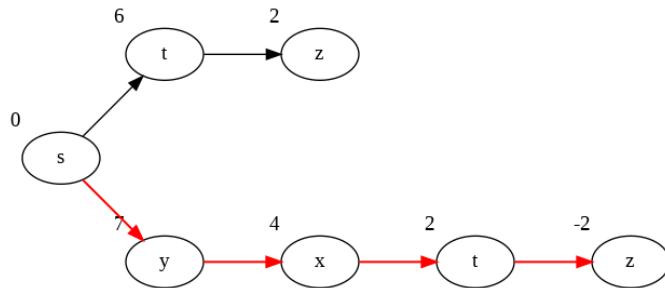


Figure 20.19: The tree structure indicates the updates on D , and the shortest path tree marked by red arrows.

Formal Bellman-Ford Algorithm In the above implementation, at each round, we made a copy of D , which is named `newD`. However, we can actually reuse the original D and update directly on it. The difference is: we would update $D[i]$ at step m with other $D[k]$ at step m instead of at step $m - 1$, making some nodes' optimal estimate even more optimal. In the previous implementation, we can guarantee that after iteration i , for all $i \in [0, n - 1]$, D_i is at most the weight of every path from s to i using at most m edges. In the new version, we end up reaching to the optimal value even earlier, but still it takes $n - 1$ passes to guarantee.

Second, the inner two `for` loops are equivalently enumerating edges: for each possible edge (k, i) , we update the best estimate for node i . With such two points modified, we get our official the Bellman-Ford algorithm, which states:

1. Initialize D and Π as D^0 and Π^0 .
2. Run a relaxation process for $|v| - 1$ passes. Within each pass, go through each edge $(u, v) \in E$, with Eq. 20.9, if using u as an intermediate node, the tentative shortest path has smaller value, update D and Π .



Implement Bellman-Ford by checking edges from adjacency list as defined in `g` for Fig. 20.12. We should notice that different ordering of vertices /edges to be relaxed leads to different intermediate results in D , though the final result is the same.

Implementation We give out an exemplary implementation

```

1 def bellman_ford(g: dict, s: str):
2     n = len(g)

```

```

3  # Assign an enumerial index for each key
4  V = g.keys()
5  # Key to index
6  ver2idx = dict(zip(V, [i for i in range(n)]))
7  # Index to key
8  idx2ver = dict(zip([i for i in range(n)], V))
9  # Initialization the dp matrix with d estimate and predecessor
10 si = ver2idx[s]
11 D = [float('inf') if i!=si else 0 for i in range(n)] # * n
12 P = [None] * n
13
14 # n-1 passes
15 for i in range(n-1):
16     # relax all edges
17     for u in V:
18         ui = ver2idx[u]
19         for v, w in g[u]:
20             vi = ver2idx[v]
21             # Update dp's minimum path value and predecessor
22             if D[vi] > D[ui] + w:
23                 D[vi] = D[ui] + w
24                 P[vi] = ui
25             print(f'D{i+1}: {D}')
26 return D, P, ver2idx, idx2ver

```

During each pass, we relax on the estimation D with the following ordering:

```

's':[( 't', 6), ('y', 7)],
't':[( 'x', 5), ('y', 8), ('z', -4)],
'x':[( 't', -2)],
'y':[( 'x', -3), ('z', 9)],
'z':[( 'x', 7)],

```

Printing out on the updates of D , we can see that it converges to the optimal value faster than the previous strict Dynamic programming version.

Time Complexity The first dynamic programming solution takes $O(|V|^3)$, and the formal Bellman-Ford takes $O(|V||E|)$. The later would be more efficient than the first if our graph is dense.

Detect Negative-weight Cycle If the graph contains no negative-weight cycle, after $|V| - 1$ passes of relaxation, D will reach to the minimum path value. Thus, if we run additional pass of relaxation, no vertex would be updated further. However, if there exists at least one negative-weight cycle, the $|V|^{th}$ update will have at least one vertex in D with decreased value.

Special Cases and Further Optimization

From the perspective of optimization, there are at least two approaches we can try to further boost the time efficiency, such as

1. special linear ordering of vertices to relax its leaving edges that leads us to its shortest-paths in just one pass of the Bellman-Ford algorithm,
2. and some greedy approach that takes only one pass of relaxation which can be similar to breath-first graph search or the Prim's algorithm.

In Fig. 20.18, suppose we are relaxing leaving edges of vertices in linear order $[s, t, y, z, x]$, the process will be as follows:

vertex	edges	relaxed vertices
s	(s, t), (s, y)	{t:6, y:7}
t	(t, x), (t, y), (t, z)	{x:11, z:2, t:6, y:7}
y	(y, x), (y, z)	{x:4, z:2, t:6, y:7}
z	(z, x)	{x:4, z:2, t:6, y:7}
x	(x, t)	{t:2, x:4, z:2, y:7}

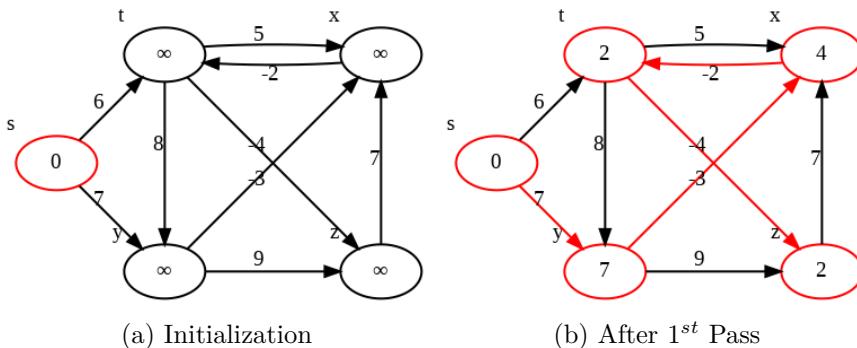


Figure 20.20: The execution of Bellman-Ford's Algorithm with ordering $[s, t, y, z, x]$.

The process is also visualized in Fig. 20.20. We see that only vertex z did not find its shortest-path weight. Why? From s to z , there are paths: $(s, t, z), (s, t, z), (s, t, y, z), (s, y, x, t, z)$. If we want to make sure after one pass of updates, vertex z reaches to its minimum shortest-path weight, we have to make sure its predecessors all reach to its minimum-path weight too which are vertex y and t . Same rule applies to its predecessors. In this graph, the ordering

vertex	predecessor
s	None
t	s, x
y	s, t
x	t, y, z
z	y, t

From the listing, we see that the pair t and x conflicts each other: t needs x as predecessor and x needs t as predecessor. Tracking down this clue, we will find out that it is due to the fact that t and x coexist in a cycle.

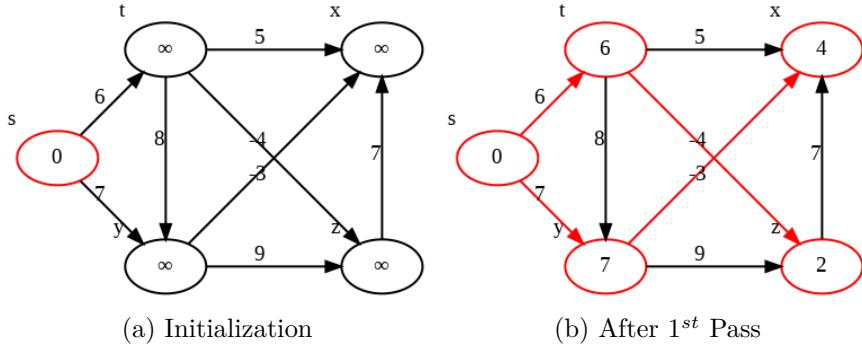


Figure 20.21: The execution of Bellman-Ford’s Algorithm on DAG using topologically sorted vertices. The red color marks the shortest-paths tree.

Order Vertices with Topological Sort Taking away edge (x, t) , we are able to obtain a topological ordering of the vertices, which is $[s, t, y, z, x]$. Relaxing vertices by this order of its leaving edges will guarantee to reach to the global-wise shortest-path weight that would otherwise be reached in $|V| - 1$ passes in Bellman-Ford algorithm using arbitrary ordering of vertices. The shortest-paths tree is shown in Fig. 20.21.

So far, we discovered a $O(|V| + |E|)$ linear algorithm for single-source shortest-path problem when the given graph being directed, weighted, and acyclic. The algorithm consists of two steps: topological sorting of vertices in G and one pass of Bellman-Ford algorithm using the reordered vertices instead of arbitrary ordering. Calling the `topo_sort` function from Section. 20.2, we have our Python code:

```

1 def bellman_ford_dag(g, s):
2     s = s
3     n = len(g)
4     # Key to index
5     ver2idx = dict(zip(g.keys(), [i for i in range(n)]))
6     # Index to key
7     idx2ver = dict(zip([i for i in range(n)], g.keys()))
8     # Convert g to index
9     ng = [[] for _ in range(n)]
10    for u in g.keys():
11        for v, _ in g[u]:
12            ui = ver2idx[u]
13            vi = ver2idx[v]
14            ng[ui].append(vi)
15    V = topo_sort(ng)
16    # Initialization the dp matrix with d estimate and predecessor
17    si = ver2idx[s]
18    dp = [(float('inf'), None) for i in range(n)]
19    dp[si] = (0, None)
20
21    # relax all edges

```

```

22     for ui in V:
23         u = idx2ver[ui]
24         for v, w in g[u]:
25             vi = ver2idx[v]
26             # Update dp's minimum path value and predecessor
27             if dp[vi][0] > dp[ui][0] + w:
28                 dp[vi] = (dp[ui][0] + w, ui)
29     return dp

```

20.5.3 Dijkstra's Algorithm

From Prim's to Dijkstra's In Breath-first Search, it hosts a FIFO queue, and whenever the vertex finishes exploring and turns into BLACK color, it is guaranteed to have the shortest-path length from the source. Similarly, in Prim's algorithm, it maintains a priority queue of cross edges between the spanning tree set S and the remaining set $V - S$, whenever a vertex is added into S , it is a part of the MST.

In the shortest-path problem, using the same initialization in Bellman-Ford algorithm, that source vertex has 0 estimate to the source and all other vertices take ∞ . Following the process of Prim's algorithm, we set a set S to save vertices that has found its shortest-path weight and predecessor, which is empty initially. Then, the algorithm starts the from the “lightest” vertex in $V - S$ to add to the set S , which is source vertex s at first, and it relax on the shortest-path estimate of vertices that are the endpoints of edges leaving the lightest vertex. This process is repeated in a loop until $V - S$ is empty. This devised approach indeed follows the principle of greedy algorithm just as Prim's algorithm does, this algorithm is called *Dijkstra's*.

How is it greedy? Dijkstra's is the “greedy” version of Bellman-ford Algorithm. At each step, dynamic programming uses Eq. 20.9 to update D_i^m by trying all possible edges that extend the paths between s and i one at a time. Bellman-ford can only guarantee to achieve the optimal solution at the very end of running all passes. However, in Dijkstra algorithm, it reaches to the optimal solution in only one step—whenever a vertex is added into S , it adds a vertex in the shortest-path tree with only “local” information.

Correctness Condition: Non-negative Weight But, how to make sure that whenever the vertex was added into set S , it reaches to its shortest-path weight? Specifically, how to ensure our locally optimal decision is global optimal? This also means after this step, no matter how many additional paths with larger path length can reach to i , they shall never have less distance. This requires all of graph edges to be non-negative.

Implementation The implementation relies on the `PriorityQueue()` customized data structure once again, where we can modify an existing item in

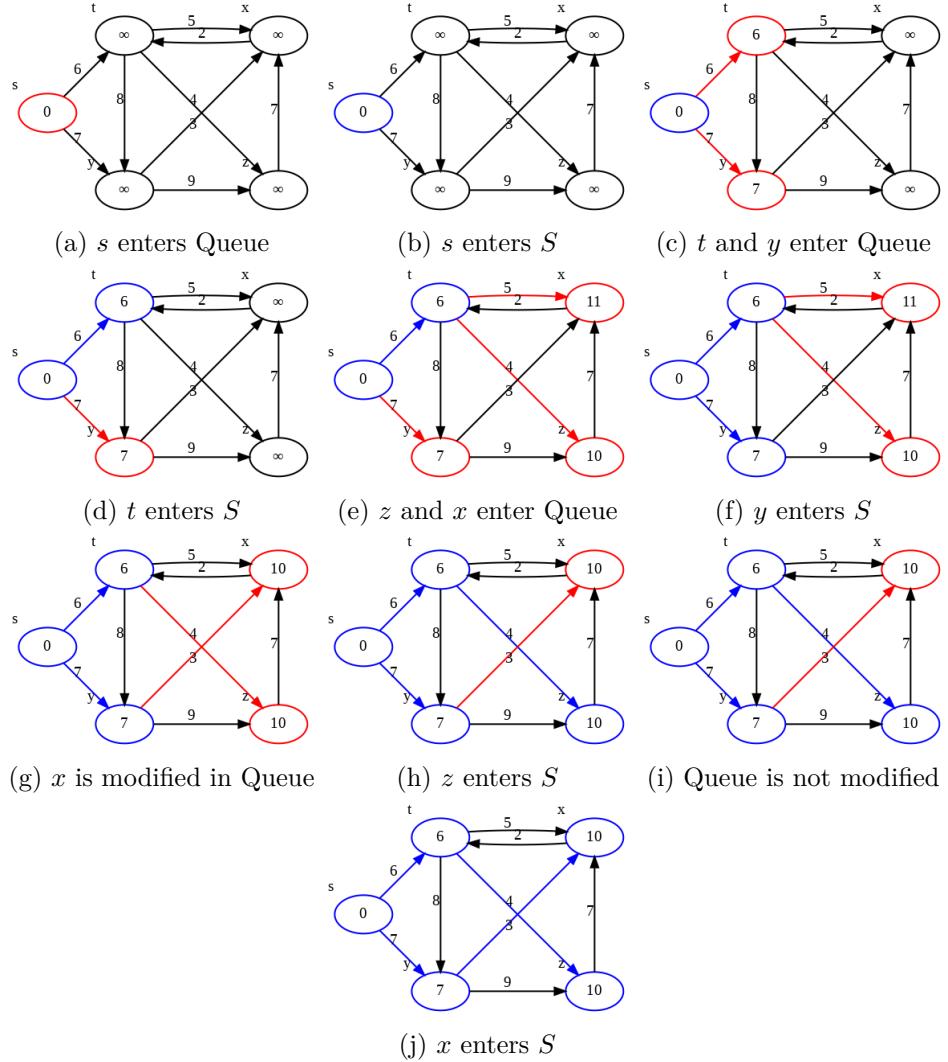


Figure 20.22: The execution of Dijkstra's Algorithm on non-negative weighted graph. Red circled vertices represent the priority queue, and blue circled vertices represent the set S . Eventually, the blue colored edges represent the shortest-paths tree.

the queue. There are two ways to apply the priority queue:

- Add all vertices into the queue all at once at the beginning. Then only `dequeue` and modification operations are needed.
- Add vertex in the queue only when it is relaxed and has a non- ∞ shortest-path estimate. The process of Dijkstra algorithm on a non-negative weighted graph that takes this approach of queue is demonstrated in Fig. 20.22 and the code is as follows:

```

1 def dijkstra(g, s):
2     Q = PriorityQueue()
3     S = []
4     # task: vertex id, priority: shortest-path estimate, info:
5     # predecessor
6     Q.add_task(task=s, priority=0, info=None)
7     visited = set()
8     while not Q.empty():
9         # Use the light vertex
10        u, up, ud = Q.pop_task()
11        visited.add(u)
12        S.append((u, ud, up))
13
14        # Relax adjacent vertices
15        for v, w in g[u]:
16            # Already found the shortest path for this id
17            if v in visited:
18                continue
19
20            vd, vp = Q.get_task(v)
21            # First time to add the task or already in the queue, but
22            # need update
23            if not vd or ud + w < vd:
24                Q.add_task(task=v, priority=ud + w, info=u)
25
26    return S

```

Complexity Analysis Once again, the complexity of Dijkstra's relies on the specific implementation of the priority queue. In our implementation, we used a customized `PriorityQueue()` which takes $|V|$ to initialize the queue. In this queue, we did not really remove the task from the queue but instead marked it as "REMOVED," so we can end up having maximum of $|E|$ vertices in the queue, making the cost of extracting the minimum item be $O(\log |E|)$. For the update, the main cost comes from inserting a new vertex through `heappush-like` operation, which is $O(\log |E|)$ too. In all, we have $|V|$ times of pops and $|V|$ times of updates, ending up with a worst-case time complexity of $O(|V| \log |E|)$.

 Try to prove the correctness of Dijkstra's using greedy algorithms' two approaches on proving.

20.5.4 All-Pairs Shortest Paths

In this section, we first summarize the solutions to single-source shortest-path problem due to the fact that the problem of finding all-pairs shortest-path problem can be naturally decomposed into $|V|$ such single sourced subproblems. Next, we systematically build into three all-pair paths algorithms we are about to learn:

Summary to Single-source Shortest-Path Algorithms The solutions vary to the type of weighted graph G that we are dealing with:

- if (1) each weight $w \in R$ and (2) only non-negative cycle, we can apply the generalist dynamic programming approach—Bellman-Ford Algorithm,
- if each weight is non-negative, i.e., $w \in R^+$, we take the greedy approach—“Dijkstra’s Algorithm”
- and (1) if the graph is acyclic and (2) only have non-negative cycles, we can run one pass of Bellman-Ford algorithm with vertices being relaxed in topologically sorted liner ordering.

Depends on which category the given graph G falls into, a naive and nature solution to all-pairs shortest-path problem can be addressed by running the corresponding algorithm $|V|$ passes—once for each vertex viewed as source in a complexity scaled by $|V|$ times.

Extended Bellman-Ford’s Algorithm

We leverage the first DP approach in Section 20.5.1. Define weight matrix W , shortest-path weight estimate matrix D , and predecessor matrix Π . We have recurrence relation:

$$D^m(i, j) = \min_{k \in [0, n-1]} (D^{m-1}(i, k) + W(k, j)), \quad (20.10)$$

$\Pi^m(i, j)$ is updated by:

$$\Pi^m(i, j) = \begin{cases} \text{None}, & \text{if } D^m(i, j) = 0 \text{ or } D^m(i, j) = \infty, \\ \operatorname{argmin}_{k \in [0, n-1]} (D^{m-1}(i, k) + W(k, j)), & \text{otherwise.} \end{cases} \quad (20.11)$$

with initialization:

$$D^0(i, j) = \begin{cases} 0, & \text{if } i = j, \\ \infty, & \text{otherwise.} \end{cases} \quad (20.12)$$

$$\Pi^0(i, j) = \text{None} \quad (20.13)$$

In detail, our extended Bellman-ford algorithm consists of these main steps:

1. Initialization: we initialize d and π using Eq. 20.12 and 20.13.
2. For every pair of vertices i and j , we update the d and π using recurrence relation in Eq. 20.10 and 20.13, respectively, for $|V| - 1$ passes.

3. Run the $|V|^{th}$ pass to decide if any negative-weight cycle exist in each rooted shortest-path tree.

To notice that after one pass of update on D since it is initialized, $D^{(1)} = W$, thus, in our implementation, only $|V| - 2$ passes of updates are needed actually. Assume we have converted the graph shown in Fig. 20.12 into a W adjacency matrix representation and a dictionary `key2idx` that maps each key to a numerical index from 0 to $|V| - 1$. This extended Bellman-ford algorithm is implemented in main function `extended_bellman_ford_with_predecessor` which calls a subfunction `bellman_ford_with_predecessor` that does one pass of relaxation and does not detect non-negative cycle. The code is as:

```

1 import copy
2 def bellman_ford_with_predecessor(W, L, P):
3     n = len(W)
4     for i in range(n): # source
5         for j in range(n): # endpoint
6             for k in range(n): # extend one edge
7                 if L[i][k] + W[k][j] < L[i][j]:
8                     L[i][j] = L[i][k] + W[k][j] # set d
9                     P[i][j] = k # set predecessor
10
11 def extended_bellman_ford_with_predecessor(W):
12     n = len(W)
13     # initialize L, first pass
14     L = copy.deepcopy(W)
15     print(f'L1 : {L} \n')
16     P = [[None for _ in range(n)] for _ in range(n)]
17     for i in range(n):
18         for j in range(n):
19             if L[i][j] != 0 and L[i][j] != float('inf'):
20                 P[i][j] = i
21     # n-2 passes
22     for i in range(n-2):
23         bellman_ford_with_predecessor(W, L, P)
24         print(f'L{i+2}: {L} \n')
25     return L, P

```

The L matrix will be having all zeros along the diagonal, in this case, it is

```
[ [0, 2, 4, 7, -2],
  [inf, 0, 3, 8, -4],
  [inf, -2, 0, 6, -6],
  [inf, -5, -3, 0, -9],
  [inf, 5, 7, 13, 0]],
```

We reconstruct the shortest-path trees and visualize them in Fig. 20.23.

Repeated Squaring Extended Bellman-Ford Algorithm

We leverage the second DP approach in Section 20.5.1. This approach bears resemblance to the repeated squaring optimization in matrix multiplication.

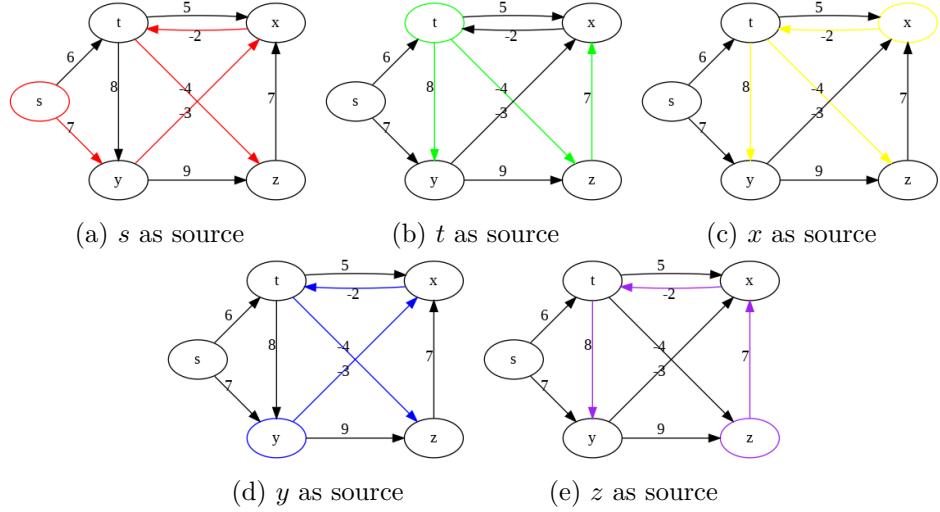


Figure 20.23: All shortest-path trees starting from each vertex.

Repeated squaring is a general method for fast computation of exponentiation with large powers of a number or more generally of a polynomial or a square matrix. The underlying algorithm design methodology is divide and conquer. Assume our input is x^n , where x is an expression, repeat squaring computes this in $O(\log n)$ steps by repeatedly squaring an intermediate result. Repeating Squaring method is actually used a lot in some advanced algorithms. Another one we will see in String algorithms.

Repeated Squaring Applied on Extended Bellman-Ford Algorithm
 If we observe the `bellman_ford_one_pass`, it has three for loops, and it shows similar pattern with matrix multiplication. Suppose A and B are both $n \times n$ matrix, and we compute $C = A \times B$, the formulation is $c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}$ which has the same pattern as of Eq. 20.10. If we use \cdot to mark `bellman_ford_one_pass` operation on L and W , we will have the following relations:

$$L^1 = L^0 \cdot W = W, \quad (20.14)$$

$$L^2 = L^1 \cdot W = W^2,$$

$$L^3 = L^2 \cdot W = W^3,$$

$$\vdots$$

$$L^{n-1} = L^{n-2} \cdot W = W^{n-1}$$

With repeated squaring technique, we can compute L^{n-1} with only

$\log(n - 1)$ round of one pass operation

$$\begin{aligned} L^1 &= W, \\ L^2 &= W \cdot W, \\ L^4 &= W^2 \cdot W^2, \\ &\vdots \end{aligned} \tag{20.15}$$

The above repetition stops when our $m \geq n - 1$. The implementation is:

```

1 import copy
2 import math
3 def bellman_ford_repeated_square(L):
4     n = len(W)
5     for i in range(n): # source
6         for j in range(n): # endpoint
7             for k in range(n): # double the extending length
8                 L[i][j] = min(L[i][j], L[i][k]+L[k][j])
9
10 def extended_bellman_ford_repeated_square(W):
11     n = len(W)
12     # initialize L, first pass
13     L = copy.deepcopy(W)
14     print(f'L1 : {L} \n')
15     # log n passes
16     for i in range(math.ceil(math.log(n))):
17         bellman_ford_repeated_square(L)
18         print(f'L{2^(i+1)}: {L} \n')
19     return L

```

The Floyd-Warshall Algorithm

We leverage the third DP approach in Section 20.5.1, this approach is called *The Floyd-Warshall Algorithm*. We directly put the code here:

```

1 def floyd_warshall(W):
2     L = copy.deepcopy(W) #L0
3     n = len(W)
4     for k in range(n): # intermediate node
5         for i in range(n): # start node
6             for j in range(n): # end node
7                 L[k][i] = min(L[k][i], L[k][j] + L[j][i])
8     return L

```


21

Advanced Data Structures

In this chapter, we extend the data structure learned from the first part with more advanced data structures. These data structures are not as widely used as the basic data structures, however, they can be often seen to implement more advanced algorithms or they can be more efficient compared with algorithms that relies on a more basic version.

21.1 Monotone Stack

A monotone Stack is a data structure the elements from the front to the end is strictly either increasing or decreasing. For example, there is a line at the hair salo, and you would naturally start from the end of the line. However, if you are allowed to kick out any person that you can win at a fight, if every one follows the rule, then the line would start with the most powerful man and end up with the weakest one. This is an example of monotonic decreasing stack.

- Monotonically Increasing Stack: to push an element e , starts from the rear element, we pop out element $r \geq e$ (violation);
- Monotonically Decreasing Stack: we pop out element $r \leq e$ (violation). T

The process of the monotone decresing stack is shown in Fig. 21.1. *Sometimes, we can relax the strict monotonic condition, and can allow the stack or queue have repeat value.*

To get the feature of the monotonic queue, with $[5, 3, 1, 2, 4]$ as example, if it is increasing:

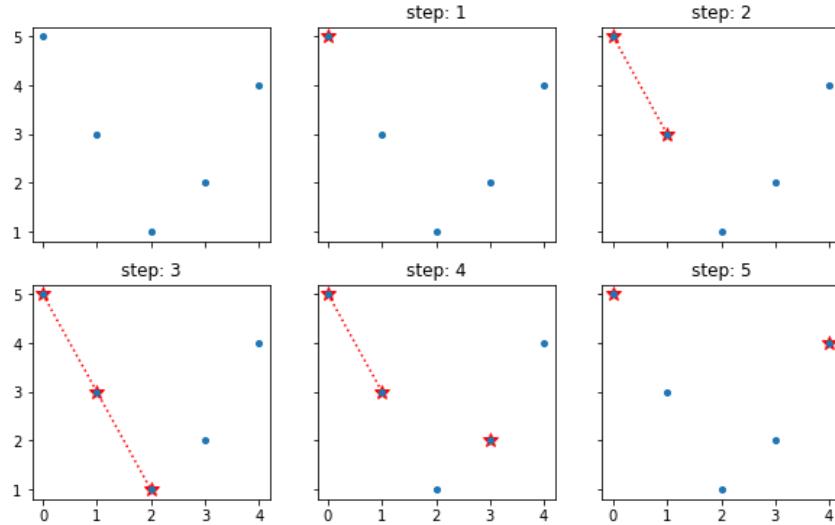


Figure 21.1: The process of decreasing monotone stack

index	v	Increasing stack	Decreasing stack
1	5	[5]	[5]
2	3	[3] 3 kick out 5	[5, 3] #3->5
3	1	[1] 1 kick out 3	[5, 3, 1] #1->3
4	2	[1, 2] #2->1	[5, 3, 2] 2 kick out 1
5	4	[1, 2, 4] #4->2	[5, 4] 4 kick out 2, 3

By observing the above process, what features we can get?

- Pushing in to get smaller/larger item to the left: When we push an element in, if there exists one element right in front of it, 1) for increasing stack, we find the **nearest smaller item to the left** of current item, 2) for decreasing stack, we find the **nearest larger item** to the left instead. In this case, we get [-1, -1, -1, 1, 2], and [-1, 5, 3, 3, 5] respectively.
- Popping out to get smaller/larger item to the right: when we pop one element out, for the kicked out item, such as in step of 2, increasing stack, 3 forced 5 to be popped out, for 5, 3 is the first smaller item to the right. Therefore, if one item is popped out, for this item, the current item that is about to be push in is 1) for increasing stack, **the nearest smaller item to its right**, 2) for decreasing stack, **the nearest larger item to its right**. In this case, we get [3, 1, -1, -1, -1], and [-1, 4, 2, 4, -1] respectively.

The conclusion is with monotone stack, we can search for smaller/larger items of current item either to its left/right.

Basic Implementation This monotonic queue is actually a data structure that needed to add/remove element from the end. In some application we might further need to remove element from the front. Thus Deque from collections fits well to implement this data structure. Now, we set up the example data:

```
1 A = [5, 3, 1, 2, 4]
2 import collections
```

Increasing Stack We can find first smaller item to left/right.

```
1 def increasingStack(A):
2     stack = collections.deque()
3     firstSmallerToLeft = [-1]*len(A)
4     firstSmallerToRight = [-1]*len(A)
5     for i,v in enumerate(A):
6         while stack and A[stack[-1]] >= v: # right is from the
7             popping out
8                 firstSmallerToRight[stack.pop()] = v # A[stack[-1]]
9                 >= v
10                if stack: #left is from the pushing in, A[stack[-1]] <
11                    v
12                    firstSmallerToLeft[i] = A[stack[-1]]
13                    stack.append(i)
14    return firstSmallerToLeft, firstSmallerToRight, stack
```

Now, run the above example with code:

```
1 firstSmallerToLeft, firstSmallerToRight, stack = increasingQueue
2     (A)
3 for i in stack:
4     print(A[i], end = ' ')
5 print('\n')
6 print(firstSmallerToLeft)
7 print(firstSmallerToRight)
```

The output is:

```
1 2 4
2
3 [-1, -1, -1, 1, 2]
4 [3, 1, -1, -1, -1]
```

Decreasing Stack We can find first larger item to left/right.

```
1 def decreasingStack(A):
2     stack = collections.deque()
3     firstLargerToLeft = [-1]*len(A)
4     firstLargerToRight = [-1]*len(A)
5     for i,v in enumerate(A):
6         while stack and A[stack[-1]] <= v:
7             firstLargerToRight[stack.pop()] = v
8
9     if stack:
```

```

10         firstLargerToLeft [ i ] = A[ stack [ -1 ] ]
11         stack .append( i )
12     return firstLargerToLeft , firstLargerToRight , stack

```

Similarly, the output is:

```

1 5 4
2
3 [-1, 5, 3, 3, 5]
4 [-1, 4, 2, 4, -1]

```

For the above problem, If we do it with brute force, then use one for loop to point at the current element, and another embedding for loop to look for the first element that is larger than current, which gives us $O(n^2)$ time complexity. If we think about the BCR, and try to trade space for efficiency, and use monotonic queue instead, we gain $O(n)$ linear time and $O(n)$ space complexity.

Monotone stack is especially useful in the problem of subarray where we need to find smaller/larger item to left/right side of an item in the array. To better understand the features and applications of monotone stack, let us look at some examples. First, we recommend the audience to practice on these obvious applications shown in LeetCode Problem Section before moving to the examples:

There is one problem that is pretty interesting:

Sliding Window Maximum/Minimum Given an array `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window. (LeetCode Probelm: 239. Sliding Window Maximum (hard))

Example :

```

Input : nums = [1,3,-1,-3,5,3,6,7], and k = 3
Output: [3,3,5,5,6,7]
Explanation :

```

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Analysis: In the process of moving the window, any item that is smaller than its predecessor will not affect the max result anymore, therefore, we can use decrease stack to remove any trough. If the window size is the same as of the array, then the maximum value is the first element in the stack

(bottom). With the sliding window, we record the max each iteration when the window size is the same as k. At each iteration, if need to remove the out of window item from the stack. For example of [5, 3, 1, 2, 4] with k = 3, we get [5, 3, 4]. At step 3, we get 5, at step 4, we remove 5 from the stack, and we get 3. At step 5, we remove 3 if it is in the stack, and we get 4. With the monotone stack, we decrease the time complexity from $O(kn)$ to $O(n)$.

```

1 import collections
2
3 def maxSlidingWindow(self, nums, k):
4     ds = collections.deque()
5     ans = []
6     for i in range(len(nums)):
7         while ds and nums[i] >= nums[ds[-1]]: indices.pop()
8         ds.append(i)
9         if i >= k - 1: ans.append(nums[ds[0]]) #append the
10            current maximum
11         if i - k + 1 == ds[0]: ds.popleft() #if the first also
the maximum number is out of window, pop it out
12     return ans

```

21.1 907. Sum of Subarray Minimums (medium). Given an array of integers A, find the sum of min(B), where B ranges over every (contiguous) subarray of A. Since the answer may be large, return the answer modulo $10^9 + 7$. Note: $1 \leq A.length \leq 30000$, $1 \leq A[i] \leq 30000$.

Example 1:

```

Input: [3,1,2,4]
Output: 17
Explanation: Subarrays are [3], [1], [2], [4], [3,1],
             [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].
Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1. Sum is 17.

```

Analysis: For this problem, using naive solution to enumerate all possible subarrays, we end up with n^2 subarray and the time complexity would be $O(n^2)$, and we will receive LTE. For this problem, we just need to sum over the minimum in each subarray. Try to consider the problem from another angle, what if we can figure out how many times each item is used as minimum value corresponding subarray? Then $res = sum(A[i]*f(i))$. If there is no duplicate in the array, then To get $f(i)$, we need to find out:

- $left[i]$, the length of strict bigger numbers on the left of $A[i]$,
- $right[i]$, the length of strict bigger numbers on the right of $A[i]$.

For the given examples, if $A[i] = 1$, then the left item is 3, and the right item is 4, we add $1 * (left_len * right_len)$ to the result. However,

if there is duplicate such as [3, 1, 4, 1], for the first 1, we need [3,1], [1], [1,4], [1, 4,1] with subbarries, and for the second 1, we need [4,1], [1] instead. Therefore, we set the right length to find the \geq item. Now, the problem is converted to the first smaller item on the left side and the first smaller or equal item on the right side. From the feature we draw above, we need to use increasing stack, as we know, from the pushing in, we find the first smaller item, and from the popping out, for the popped out item, the current item is the first smaller item on the right side. The code is as:

```

1 def sumSubarrayMins(self, A):
2     n, mod = len(A), 10**9 + 7
3     left, s1 = [1] * n, []
4     right = [n-i for i in range(n)]
5     for i in range(n): # find first smaller to the left
6         from pushing in
7             while s1 and A[s1[-1]] > A[i]: # can be equal
8                 index = s1.pop()
9                 right[index] = i-index # kicked out
10            if s1:
11                left[i] = i-s1[-1]
12            else:
13                left[i] = i+1
14            s1.append(i)
15     return sum(a * l * r for a, l, r in zip(A, left, right))
) % mod

```

The above code, we can do a simple improvement, by adding 0 to each side of the array. Then eventually there will only have [0, 0] in the stack. All of the items originally in the array they will be popped out, each popping, we can sum up the result directly:

```

1 def sumSubarrayMins(self, A):
2     res = 0
3     s = []
4     A = [0] + A + [0]
5     for i, x in enumerate(A):
6         while s and A[s[-1]] > x:
7             j = s.pop()
8             k = s[-1]
9             res += A[j] * (i - j) * (j - k)
10            s.append(i)
11     return res % (10**9 + 7)

```

21.2 Disjoint Set

Disjoint-set data structure (aka union-find data structure or merge-find set) maintains a collection $S = \{S_1, S_2, \dots, S_k\}$ of disjoint *dynamic* sets by partitioning a set of elements. We identify each set by a **representative**, which is some member of the set. It does matter which member is used

only if we get the same answer both times if we ask for the representative twice without modifying the set. Choosing the smallest member in a set as representative is an exemplary prespecified rule. According to its typical applications such as implementing Kruskal's minimum spanning tree algorithm and tracking connected components dynamically, disjoint-set should support the following operations:

1. `make_set(x)`: create a new set whose only member is x . To keep these sets to be disjoint, this member should not already be in some existent sets.
2. `union(x, y)`: unites the two dynamic sets that contain x and y , say $S_x \cup S_y$ into a new set that is the union of these two sets. In practice, we merge one set into the other say S_y into S_x , we then remove/destroy S_y . This will be more efficient than create a new one that unions and destroy the other two.
3. `find_set(x)`: returns a pointer to the representative of the set that contains x .

Applications Disjoint sets are applied to implement union-find algorithm where performs `find_set` and `union`. Union-find algorithms can be used into some basic graph algorithms, such as cycle detection, tracking connected components in the graph dynamically,¹, Krauskal's MST algorithm, and Dijkstra's Shortest path algorithm.

Connected Component Before we move to the implementation, let us first see how disjoint set can be applied to connected components. At first, we assign a set id for each vertex in the graph. Then we traverse each edge, and if the two endpoints of the edge belongs to different set, then we union the two sets. As shown in the process, first vertex 0 and 1 has different set id, then we update 1's id to 0. For edge (1, 2), we update 2's id to 0. For edge(0, 2), they are already in the same set, no update needed. We apply the same process with edge (2, 4), (3, 4), and (5, 6).

21.2.1 Basic Implementation with Linked-list or List

Before we head off to more efficient and complex implementation, we first implement a baseline for the convenience of comparison. The key for the implementation is two dictionaries named `item_set` (saves the mapping between item and its set id, which will only be one to one) and `set_item` (the value of the key will be a list, because one set will have one to multiple relation).

¹where new edge will be added and the search based algorithm each time will be rerun to find them again

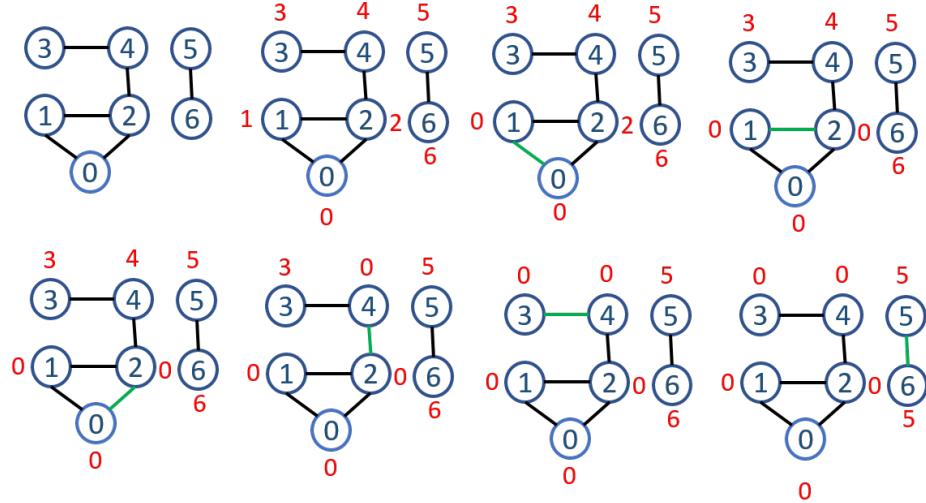


Figure 21.2: The connected components using disjoint set.

If our coding is right, each item must have an item when `find_set` function is called, if not we will call `make_set`. For each existing `set`, it will have at least one item. For function `union`, we choose the set that has less items to merge to the one that with more items.

```
1 class DisjointSet():
2     '''Implement a basic disjoint set'''
3     def __init__(self, items):
4         self.n = len(items)
5         self.item_set = dict(zip(items, [i for i in range(self.n)]))
6         # first each set only has one item [i], this can be one->
7         # multiple match
8         self.set_item = dict(zip([i for i in range(self.n)], [[item]
9             for item in items])) # each item will always belong to one
10        set
11
12    def make_set(self, item):
13        '''make set for new incoming set'''
14        if item in self.item_set:
15            return
16
17        self.item_set[item] = self.n
18        self.n += 1
19
20    def find_set(self, item):
21        if item in self.item_set:
22            return self.item_set[item]
23        else:
24            print('not in the set yet: ', item)
25            return None
```

```

23     def union(self, x, y):
24         id_x = self.find_set(x)
25         id_y = self.find_set(y)
26         if id_x == id_y:
27             return
28
29         sid, lid = id_x, id_y
30         if len(self.set_item[id_x]) > len(self.set_item[id_y]):
31             sid, lid = id_y, id_x
32         # merge items in sid to lid
33         for item in self.set_item[sid]:
34             self.item_set[item] = lid
35             self.set_item[lid] += self.set_item[sid]
36         del self.set_item[sid]
37     return

```

Complexity For n items, we spend $O(n)$ time to initialize the two hashmaps. With the help of hashmap, function `find_set` tasks only $O(1)$ time, accumulating it will give us $O(n)$. For function `union`, it takes more effort to analyze. From another angle, for one item x , it will only update its item id when we are unioning it to another set x_1 . The first time, the resulting set x_1 will have at least two items. The second update will be union x_1 to x_2 . Because the merged one will have smaller length, thus the resulting items in x_2 will at least be 4. Then it is the third, ..., up to k updates. Because a resulting set will at most has n in size, so for each item, at most $\log n$ updates will be needed. For n items, this makes the upper bound for `union` to be $n \log n$.

However, for our implementation, we has additional cost, which is in `union`, where we merge the list. This cost can be easily limited to constant by using linked list. However, even with `list`, there are different ways to concatenate one list to another:

1. Use `+` operator: The time complexity of the concat operation for two lists, A and B, is $O(A + B)$. This is because you aren't adding to one list, but instead are creating a whole new list and populating it with elements from both A and B, requiring you to iterate through both.
2. `extend(lst)`: Use `extend` which doesn't create a new list but adds to the original. The time complexity should only be $O(1)$. On the other hand `l += [i]` modifies the original list and behaves like `extend`.

21.2.2 Implementation with Disjoint-set Forests

Instead of using linear linked list, we use tree structure. Different with trees we have introduced before that a node points to its children, an item here will only points to its parent. A tree represents a set, and the root node is the representative and it points to itself. The straightforward algorithms

that use this structure are not faster than the linked-list version. By introducing two heuristics—“Union by rank” and “path compression”—we can achieve asymptotically optimal disjoint-set data structure.

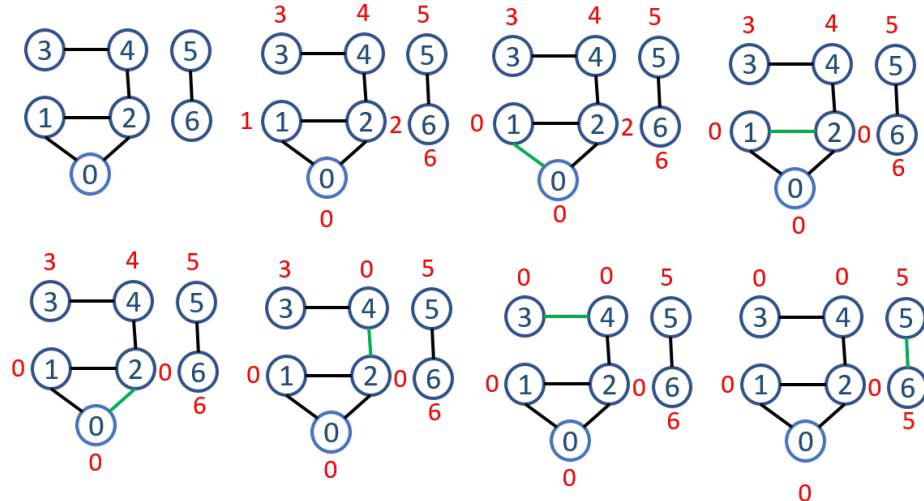


Figure 21.3: A disjoint forest

Naive Version

We first need to create a `Node` class which stores `item` and another parent pointer `parent`. An `item` can be any immutable data structure with necessary information represents a node.

```

1 class Node:
2     def __init__(self, item):
3         self.item = item # save node information
4         self.parent = None

```

We need one dict data structure `item_finder` and one set data structure `sets` to track nodes and set. From `item_finder` we can do `(item, node)` map to find node, and then from the node further we can find its set representative node or execute `union` operation. `sets` is used to track all the representative nodes. When we union two sets, the one merged to the other will be deleted in `sets`. At the easy version, `make_set` will create tree with only one node. `find_set` will start from the node and traverse all the way back to its final parent which is when `node.parent==node`. And a `union` operation will simply point one tree's root node to the root of another through `parent`. The code is as follows:

```

1 class DisjointSet():
2     '''Implement with disjoint-set forest'''
3     def __init__(self, items):

```

```

4     self.n = len(items)
5     self.item_finder = dict()
6     self.sets = set() # sets will have only the parent node
7
8     for item in items:
9         node = Node(item)
10        node.parent = node
11        self.item_finder[item] = node # from item we can find the
12        node
13        self.sets.add(node)
14
15    def make_set(self, item):
16        '''make set for new incoming set'''
17        if item in self.item_finder:
18            return
19
20        node = Node(item)
21        node.parent = node
22        self.item_finder[item] = node
23        self.sets.add(node)
24        self.n += 1
25
26    def find_set(self, item):
27        # from item->node->parent to set representative
28        if item not in self.item_finder:
29            print('not in the set yet: ', item)
30            return None
31        node = self.item_finder[item]
32        while node.parent != node:
33            node = node.parent
34        return node
35
36    def union(self, x, y):
37        node_x = self.find_set(x)
38        node_y = self.find_set(y)
39        if node_x.item == node_y.item:
40            return
41
42        #the root of one tree to point to the root of the other
43        # merge x to y
44        node_x.parent = node_y
45        #remove one set
46        self.sets.remove(node_x)
47        return
48
49    def __str__(self):
50        ans = ''
51        for root in self.sets:
52            ans += 'set: ' + str(root.item) + '\n'
53
54    def print_set(self, item):
55        if item in self.item_finder:
56            node = self.item_finder[item]
```

```

57     print(node.item, '→', end=' ')
58     while node.parent != node:
59         node = node.parent
60         print(node.item, '→', end=' ')

```

Let's run an example:

```

1 ds = DisjointSet(items=[i for i in range(5)])
2 ds.union(0,1)
3 ds.union(1,2)
4 ds.union(2,3)
5 ds.union(3,4)
6 print(ds)
7 for item in ds.item_finder.keys():
8     ds.print_set(item)
9     print(' ')

```

The output is:

```

set: 4

0 →1 →2 →3 →4 →
1 →2 →3 →4 →
2 →3 →4 →
3 →4 →
4 →

```

The above implementation, both `make_set` and `union` takes $O(1)$ time complexity. The main time complexity is incurred at `find_set`, which traverse a path from node to root. If we assume each tree in the disjoint-set forest is balanced, the upper bound of this operation will be $O(\log n)$. However, if the tree is as worse as a linear linked list, the time complexity will goes to $O(n)$. This makes the total time complexity from $O(n \log n)$ to $O(n^2)$.

Heuristics

Union by Rank As we have seen from the above example, A sequence of $n - 1$ `union` operations may create a tree that is just a linear chain of n nodes. Union by rank, which is similar to the weighted-union heuristic we used with the linked list implementation, is applied to avoid the worst case. For each node, other than the parent pointer, it adds `rank` to track the upper bound of the height of the associated node (the number of edges in the longest simple path between the node and a descendant leaf). In union by rank, we make the root with smaller rank point to the root with larger rank.

In the initialization, and `make_set` operation, a single noded tree has an initial rank of 0. In `union(x, y)`, there will exist three cases:

```

Case 1 x.rank == y.rank:
    join x to y
    y.rank += 1
Case 2: x.rank < y.rank:

```

```

        join y to x
        x.rank += 1
Case 3: x.rank > y.rank:
        join y to x
        x's rank stay unchanged

```

Now, with adding `rank` to the node. We modify the naive implementation:

```

1 class Node:
2     def __init__(self, item):
3         self.item = item # save node information
4         self.parent = None
5         self.rank = 0

```

The updated implementation of `union`:

```

1 def union(self, x, y):
2     node_x = self.find_set(x)
3     node_y = self.find_set(y)
4     if node_x.item == node_y.item:
5         return
6
7     # link
8     if node_x.rank > node_y.rank:
9         node_y.parent = node_x
10        #remove one set
11        self.sets.remove(node_y)
12    elif node_x.rank < node_y.rank:
13        node_x.parent = node_y
14        self.sets.remove(node_x)
15    else:
16        node_x.parent = node_y
17        node_y.rank += 1
18        self.sets.remove(node_x)
19

```

Path Compression In our naive implementation, `find_set` took the most time. With path compression, during the process of `find_set`, it simply make each node on the find path point directly to its root. Path Compression wont affect the rank of each node. Now, we modify this function:

```

1 def _find_parent(self, node):
2     while node.parent != node:
3         node = node.parent
4     return node
5
6 def find_set(self, item):
7     '''modified to do path compression'''
8     # from item->node->parent to set representative
9     if item not in self.item_finder:
10        print('not in the set yet: ', item)
11        return None
12     node = self.item_finder[item]

```

```

13     node.parent = self._find_parent(node) # change node's parent
14         to the root node
15     return node.parent

```

The same example, the output will be:

```

set: 1
0 ->1 ->
1 ->
2 ->1 ->
3 ->1 ->
4 ->1 ->

```

```

1 import time, random
2 t0 = time.time()
3 n = 100000
4 ds = DisjointSet(items=[i for i in range(n)])
5 for _ in range(n):
6     i, j = random.randint(0, n-1), random.randint(0, n-1) #[0,n]
7     ds.union(i, j)
8 print('time: ', time.time()-t0)

```

Experiment to the running time of Linked-list VS naive forest VS heuristic forest

We run the disjoint set with $n=100,000$, and with n times of union:

```

1 import time, random
2 t0 = time.time()
3 n = 100000
4 ds = DisjointSet(items=[i for i in range(n)])
5 for _ in range(n):
6     i, j = random.randint(0, n-1), random.randint(0, n-1) #[0,
7         n]
8     ds.union(i, j)
8 print('time: ', time.time()-t0)

```

The resulting time is: 1.09s, 50.4s, 1.19s

Note As we see, in our implementation, we have never removed any item from disjoint-set structure. Also, from the above implementation, we know the sets of the nodes, but we can't track items from the root node. How can we further improve this?

21.3 Fibonacci Heap

21.4 Exercises

21.4.1 Knowledge Check

21.4.2 Coding Practice

Disjoint Set

1. 305. Number of Islands II (hard)

22

String Pattern Matching Algorithms

Pattern matching is a fundamental string processing problem. Pattern matching algorithms are also called string searching algorithms, and it is defined a class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text. Based on if some mismatches are allowed or not, we have **Exact or Approximate** Pattern Matching. In this section, we start from exact single-pattern matching algorithms where we only need to find one pattern in a given string or text. Based on how many patterns we might have, we have **one-time or multiple-times** string pattern matching problems. For multiple-times matching, preprocessing the text using suffix array/trie/tree can improve the total efficiency. This chapter is organized as:

1. Exact Pattern Matching: includes one-pattern and multiple patterns.
2. Approximate Pattern Matching:

22.1 Exact Single-Pattern Matching

Exact Single-pattern Matching Problem Given two strings or two arrays, one is pattern P which has size m , and the other is the target string or text T which has size n , the exact single-pattern matching problem is defined as finding the first one or all occurrences of pattern P in the T as substring, and return the starting indexes of all the occurrences.

Brute Force Solution The naive searching is straightforward, we slide the pattern P like sliding window algorithm through the text T one by one item. At each position i , we compare P with $T[i:i+m]$. In this process, we

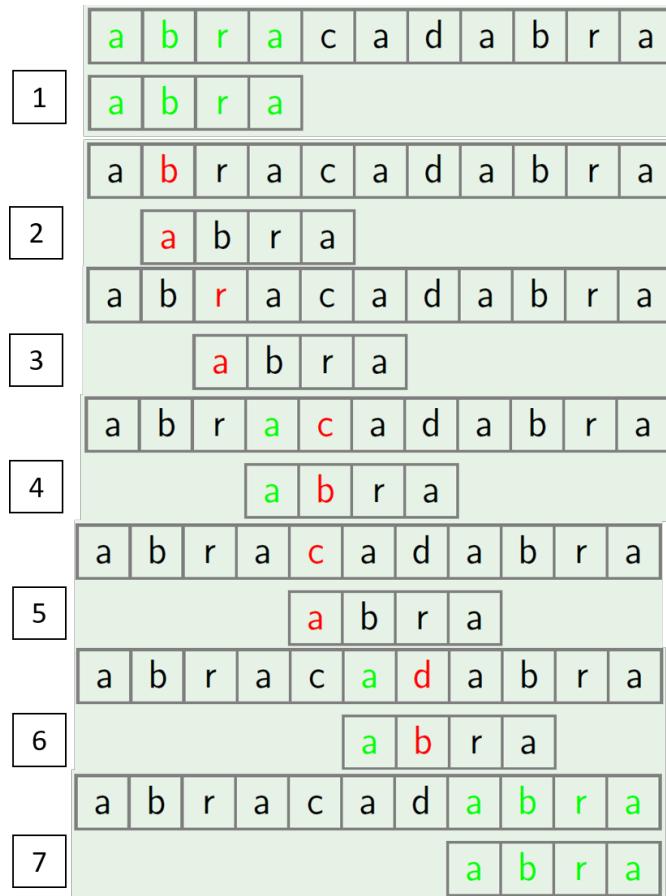


Figure 22.1: The process of the brute force exact pattern matching

need to do $n - m$ times of comparison, and each comparison takes maximum of m times of computation. This brute force solution gives $O(mn)$ time complexity.

```

1 def bruteForcePatternMatching(p, s):
2     if len(p) > len(s):
3         return [-1]
4     m, n = len(p), len(s)
5     ans = []
6     for i in range(n-m+1):
7         if s[i:i+m] == p:
8             ans.append(i)
9     return ans
10
11 p = "AABA"
12 s = "AABAACAADAABAABA"
13 print(bruteForcePatternMatching(p,s))
14 # output
15 # [0, 9, 12]
```

We write it in another way that use less built-in python function:

```

1 def bruteForcePatternMatchingAll(p, s):
2     if not s or not p:
3         return []
4     m, n = len(p), len(s)
5     i, j = 0, 0
6     ans = []
7     while i < n:
8         # do the pattern matching
9         if s[i] == p[j]:
10             i += 1
11             j += 1
12             if j == m: #collect position
13                 ans.append(i - j)
14                 i = i - j + 1
15                 j = 0
16         else:
17             i = i - j + 1
18             j = 0
19     return ans

```

For LeetCode Problems, most times, brute force solution will not be accepted and receive LTE. In real applications, such as human genome matching, the text can have approximate size of $3 * 10^9$ and the pattern can be very long to, such as 10^8 . Therefore, other faster algorithms are needed to improve the efficiency.

The other algorithms requires us preprocess either/both the pattern and text. In this book, we mainly discuss three algorithms:

1. Knuth Morris Pratt (KMP) Algorithm (Section 22.1.1). KMP is a linear algorithm, and it should mostly be enough to solve interview related string matching, and also once we understand the algorithm, the implementation is quite trivial, which makes it a very good algorithm during interviews. It has $O(m + n)$ and $O(m)$ in the case of the time and space complexity.
2. Suffix Trie/Tree/Array Matching (Section 22.2.2).

22.1.1 Prefix Function and Knuth Morris Pratt (KMP)

In the above brute force solution, we compare our pattern with each item as starting window in the text. Each matching result is independent of each other, which is a lot of information lose to improve the efficiency.

Skipping Positions See Fig. 22.1, we know a matching at step 1. Is it necessary for us to do step 2 and step 3? The pattern itself tells us it is impossible to get a match at step 2 and step 3 because 'b' will mismatch 'a' and 'i' will mismatch 'a' too. However, at the original step 4, by analyzing

the pattern itself we know 'a' will match 'a', and any step further, we have not enough information to cover, therefore, step 4 is necessary to compare 'c' with 'b' in the pattern. In this example, step 4, 5, 6, 7 are all needed but step 4, 5, 6 will only end up do one or two comparison each step.

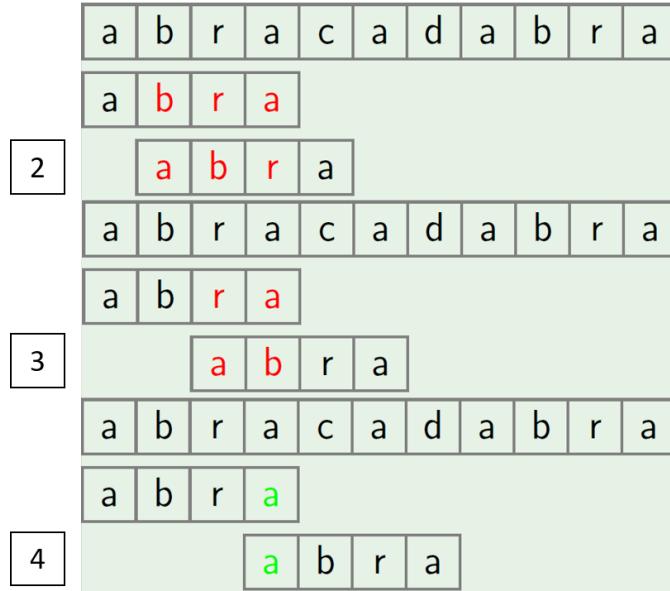


Figure 22.2: The Skipping Rule

The reason why step 2 and 3 can be skipped can be shown from Fig. 22.2. If we analyze our pattern at first, we will know at step 2 and step 3, "bra" not equals to "abr" and "ra" not equals to "ab". While at step 4, we do have "a" equals to "a". If we observe further of the relations of these pairs, we will know they are suffix and prefix of the same length of the pattern. Inspired by this, we define **border** of string S as a prefix of S which is equals to a suffix of the same length of S, but not equals to the whole S. For example:

```
'a' is a border of 'bara'
'ab' is a border of 'abcdab'
'ab' is not a border of 'ab'
```

Prefix Function A Prefix function for a string P generates an array l (lps is short for failure loopkp table) of the same length of string, where $lps[i]$ is the length of the longest border of for prefix substring $P[0...i]$. Mathematically the definition of prefix function can be written as follows:

$$l[i] = \max_{k=0,\dots,i} \{k : P[0\dots k-1] = P[i-(k-1)\dots i]\} \quad (22.1)$$

The naive implementation of prefix-function takes $O(n^3)$:

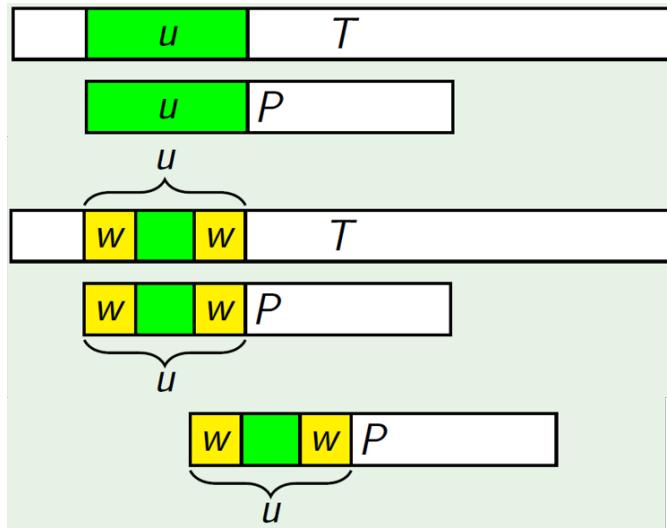


Figure 22.3: The Sliding Rule

```

1 def naiveLps(p: str):
2     dp = [0] * len(p)
3     for i in range(1, len(p)):
4         for l in range(i, 0, -1): # from maximum length to length 1
5             prefix = p[0: l]
6             suffix = p[i - l + 1: i + 1]
7             #print(prefix, suffix)
8             if prefix == suffix:
9                 dp[i] = 1
10                break
11    return dp

```

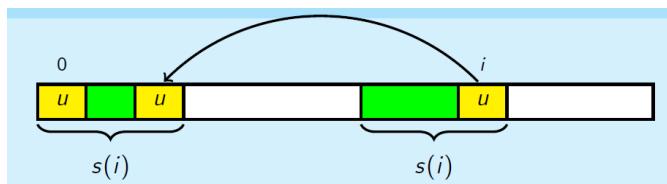


Figure 22.4: Proof of Lemma

For example, prefix function of string “abcabcd” is [0,0,0,1,2,3,0]. The trivial algorithm to implement this has $O(n^3)$ time complexity (one for loop for i , second nested for loop for k , and another n for comparing corresponding substring), which exactly follows the definition of the prefix function. The efficient algorithm which is demonstrated to run in $O(n)$ was proposed by Knuth and Pratt and independently from them by Morris in 1977. It was used as the main function of a substring search algorithm. This is the core of Knuth Morris Pratt (KMP) algorithm. In order to implement the prefix

function in linear time, we first need to utilize two properties (facts) for the purpose of two further optimization:

1. Observation: $\pi[i + 1] \leq \pi[i] + 1$, which states that the value of the prefix function can either increase by one, stay the same, or decrease by some amount.
2. Lemma: **If $l[i] > 0$, then all borders of $P[0...i]$ but for the longest one are also borders of $P[0...l(i) - 1]$.** The proof is: As shown in Fig. 22.4, $l(i)$ is the longest border for $P[0...i]$. We let μ be another shorter border of $P[0...i]$ such that $|\mu| < l(i)$. Because the first $l(i)$ and the second is the same, this means at the first $l(i)$, the suffix of $l(i)$ that of the same length of μ is μ . This states that μ is both a border of $P[0...l(i)-1]$.

Now, with such knowledge we can do the following two further optimization:

1. With 1, the complexity can be reduced to $O(n^2)$ by getting rid of the for loop on k . Because each step the prefix function can grow at most one. And among all iterations of i , it can grow at most n steps, and also only can decrease a total of n steps.
2. With 2, we can further get rid of the $O(n)$ string comparison each step. To accomplish this, we have to use all the information computed in the previous steps: all borders of $P[0...i]$ (assuming it has k in total) can be enumerated from the longest to shortest as: $b_0 = \pi(i)$, $b_1 = \pi(b_0 - 1)$, ..., $b_{k-1} = \pi(b_{k-2} - 1)$ ($b_{k-1} = 0$). Therefore, at step posited at $i + 1$, instead of comparing string $s[0...\pi(i)]$ with $s[i - (\pi(i) - 1)...i]$, comparison of char $s[\pi(i)]$ and $s[i]$ is needed.

Implementation of Prefix Function for a Given String S Let's recap the above optimization to get the final algorithm which computes prefix function in $O(n)$. This step is of key importance to the success of KMP algorithm. Let's understand this together with the algorithm statement and the code.

1. Initialization: assign n space to l array and set $l_0 = 0$.
2. A for loop in range of $[1, m-1]$ to compute $l(i)$. Set a variable $j = l(i - 1)$, and a while loop over j until $j = 0$: check if $s[j] == s[i]$; if true, $l(i) = j + 1$, otherwise reassign $j = l(j - 1)$ in order to check smaller border.

```

1 def prefix_function(s):
2     n = len(s)
3     pi = [0] * n

```

```

4   for i in range(1, n):
5       # compute l(i)
6       j = pi[i-1]
7       while j > 0 and s[i] != s[j]: # try all borders of s
8           [0...i-1], from the longest to the shortest
9           j = pi[j-1]
10      # check the character
11      if s[i] == s[j]:
12          pi[i] = j + 1
13
14  return pi

```

Run an example:

```

1 S = 'abcabcd'
2 print('The prefix function of: ', S, " is ", prefix_function(S))
3
4 The prefix function of: abcabcd  is  [0, 0, 0, 1, 2, 3, 0]

```

Knuth Morris Pratt (KMP) Back to the problem of exact pattern matching, we first build a new string as $s = P + \$ + T$, which is a concatenation of pattern P, '\$', and text T. Let us calculate the prefix function of string s. Now, let us think about the meaning of the prefix function, except for the first $m + 1$ items (which belong to the string P and the separator '\$'):

1. For all i , $\pi[i] \leq m$ because of the separator '\$' in the middle of the pattern and the text that acts as a separator.
2. If $\pi[i] = m$, i.e. $K[0 : m] = K[i - m : i] = P$. This means that the pattern P appears completely in the new string s and ends at position i . Now, we convert i to the starting position of pattern in T with $i - 2m$.
3. If $f[i] < m$, no full occurrence of pattern ends with position i.

Thus the Knuth-Morris-Pratt algorithm solves the problem in $O(n + m)$ time and $O(n + m)$ memory. And can be simply implemented with prefix function as follows:

```

1 def KMP_coarse(p, t):
2     m = len(p)
3     s = p + '$' + t
4     n = len(s)
5     pi = prefix_function(s)
6     ans = []
7     for i in range(2*m, n):
8         if pi[i] == m:
9             ans.append(i - 2*m)
10    return ans

```

Because for all $\pi[i] \leq m$: for i in $[0, m-1]$, we save the border in π ; for i in $[m, n+m-1]$, we set up a global variable j to track the last border. We can decrease the space complexity in $O(m)$. The Python implementation is given as:

```

1 def KMP(p, t):
2     m = len(p)
3     s = p + '$' + t
4     n = len(s)
5     pi = [0] * m
6     j = pi[0]
7     ans = []
8     for i in range(1, n):
9         # compute l(i)
10        while j > 0 and s[i] != s[j]: # try all borders of s
11            j = pi[j-1]
12        # check the character
13        if s[i] == s[j]:
14            j += 1
15        # record the result
16        if j == m:
17            ans.append(i-2*m)
18        # save the result if i in [0, m-1]
19        if i < m:
20            pi[i] = j
21 return ans

```

Run an example:

```

1 t = 'textbooktext'
2 p = 'text'
3 print(KMP(p, t))
4 # output
5 # [0, 8]

```

Sliding Rule with Border Information Now, assuming we know how to compute the border information, how do we slide instead compared with the brute force solution? There are three steps, with Fig. 22.3 as demonstration:

1. Find longest common prefix μ .
2. Find w – the longest border of μ .
3. Move P such that prefix w in P aligns with suffix w of μ in T.

Knuth Morris Pratt $O(m + n)$ Now, to complete the picture of KMP, when we have the lookup table at hand, when we failed to match i and j , we set $j = \text{lps}[j-1]$, and i doest not need to backtrack.

```

1 def KMP(p, ps):
2     f = LPS(p)
3     n = m, n = len(p), len(ts)
4
5     i = 0 # index in s
6     j = 0 # index in p
7     pos = []
8     while i < n:
9         if p[j] == s[i]:
10             i += 1
11             posj += 1
12             dp[i] = pos
13             if dp[i] == m: if j == m: # i at i+1, j at f[j-1]
14                 print("Found pattern at index ", i-j)
15                 ans.append(i-2*mj)
16             i += 1
17         else:
18             if pos > 0: j = f[j-1]
19             else: # mismatch at i and j
20                 if j != 0: # if j can retreat with lps, then i keep
21                     the same
22                     pos = dp[posj = f[j-1]]
23                 else:
24                     i += 1 #the value is 0
25             return ans # if j needs to start over, i moves too
26             i += 1
27     return ans
28 print(KMP(p,s))
# [0, 9, 12]

```

22.1.2 More Applications of Prefix Functions

Counting the number of occurrences of each prefix

Counting the number of occurrences of different substring in a string

Compressing a string

22.1.3 Z-function

Definition and Implementation

Z-function for a string s of length n is defined as an array $z[i] = k, i \in [1, n - 1]$. At item $z[i] = k$ stores the longest substring starting at index i which is also a prefix of string s . To notice, the length of the substring has to be smaller than the whole length, therefore, $z[0] = 0$. In other words, it means the the length of the longest common prefix between s and substring $s[i : n]$. For example:

```
"aaaaa" - [0,4,3,2,1]
a
a substring 'aaaa' = prefix 'aaaa'
a substring 'aaa' = prefix 'aaa'
a substring 'aa' = prefix 'aa'
a substring 'a' = prefix 'a'
```

Another Example.

```
"aaabaab" - [0,2,1,0,2,1,0]
a 0
a  substring 'aa' = prefix 'aa'
a  substring 'a' = prefix 'a'
b 0
a  substring 'aa' = prefix 'aa'
a  substring 'a' = prefix 'a'
b
```

z-function can be represented with a formula:

$$l[i] = \max_{k=0,\dots,i} \{k + 1 : P[0\dots k] = P[i\dots i+k]\} \quad (22.2)$$

The naive implementation of z-function takes $O(n^2)$ time complexity just as the prefix function.

```
1 def naiveZF(s):
2     n = len(s)
3     z = [0] * n
4     for i in range(1, n): # starting point
5         k = 0
6         while i + k < n and s[i + k] == s[k]:
7             k += 1
8         z[i] = k
9     return z
```

Z-function Property Here, we show how we can implement it in $O(n)$. To compute $z[i]$, do we have to start at i , then follows the order of $i+1, i+2, \dots, i+k$? The answer is No. First, As shown in Fig. 22.5, for a given

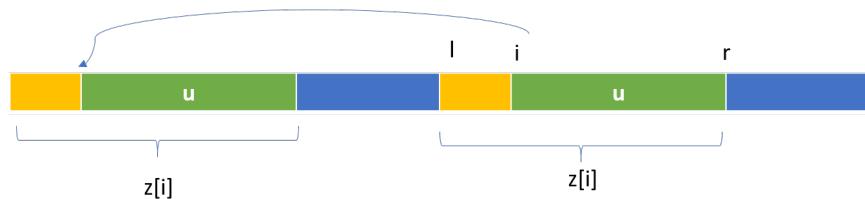


Figure 22.5: Z function property

position i , $[l, r]$ is one of its preceding non-zero $z[p], p < i$, which has the furthest right boundary r . We can think it as a rightmost window, wherein $s[l, r] = s[0, r - l + 1]$. $s[0, i - l]$ is marked as yellow. We divide the area in

range $[0, r - l + 1]$ into yellow $[0, l - i]$ and a green parts $[l - i + 1, r - l + 1]$. Therefore, to compare range $[i, r]$ with prefix is the same as of comparing range $[l - i + 1, r - l + 1]$ with the prefix, which already has a result $z[i - l]$. So instead, our k can start from position $z[i - l]$. However, there are two more restrictions:

1. Enable to utilize z-function property, $r \geq i$ because the index r can be seen as “boundary” to which our string s has been scanned by the algorithm.
2. The initial approximation for $z[i]$ is bounded by the length between r and i , which is $r - i + 1$. Therefore, we modify our initial approximation to $z[i]$ to $z[i] = \min(r - i + 1, z[i - l])$ instead.

Now, the $O(n)$ implementation is given as follows:

```

1 def linearZF(s):
2     n = len(s)
3     z = [0] * n
4     l = r = 0
5     for i in range(1, n):
6         k = 0
7         if i <= r: # r is the right bound has been scanned
8             k = min(r-i+1, z[i-1])
9         while i + k < n and s[i+k] == s[k]:
10             k += 1
11         # update the boundary
12         if i + k - 1 > r:
13             l = i
14             r = i + k - 1
15             z[i] = k
16     return z

```

Applications

The applications of Z-function are largely similar to those of prefix function. Therefore, the applications will be explained briefly compared with the applications of prefix functions. If you have problems to understand this section, please read the prefix function first.

Exact Single-Pattern Matching In this problem set, we are asked to find all occurrences of the pattern p inside the text t . We can do the same as of in the KMP, we create a new string $s = p + \$ + t$. Then, we compute the z-function for s . With the z array, for $z[i] = k$, if $k = |p|$, then we know there is one occurrence of p starting in the i -th position in s , which is $i - (|p| + 1)$ in the t .

```

1 def findPattern(p, t):
2     s = p + '$' + t

```

```

3   m = len(p)
4   z = (linearZF(s))
5   ans = []
6   for i, v in enumerate(z):
7       if v == m:
8           ans.append(i-m-1)
9   return ans

```

Number of distinct substrings in a string Given a string s of length n , count the number of distinct substrings of s .

To solve this problem we need to use dynamic programming and the subproblems are $s[0\dots 0]$, $s[0\dots 1]$, ..., $s[1\dots i]$, ..., $s[0\dots n-1]$. For example, given “abc”,

```

subproblem 1: 'a', dp[0] = 1
subproblem 2: 'ab', dp[1] = 2, with new substrings 'b', 'ab'
subproblem 3, 'abc', dp[2] = 3, new substrs 'c', 'bc', 'abc'

```

We know the maximum for $dp[i]$ is $i + 1$, however for cases like “aaa”, the situation is different:

```

subproblem 1: 'a', dp[0] = 1
subproblem 2: 'aa', dp[1] = 1, 'aa', because 'a_1 == 'a_0'
subproblem 3, 'aaa', dp[2] = 1, new substrs 'aaa', because '
    a_0a_1='a_1a_2', 'a_2' = 'a_0'.

```

If for each subproblem i , we take the string $s[0\dots i]$ and reverse it $i\dots 0$. If using z-function on this substring, we can find the number of prefixes of the reversed string are found somewhere else in it, which is the maximum value of its z-function. This is because if we know $z[j] = \max(k)$, then $s[i\dots i+\max-1] = s[i-j\dots i-j+\max]$, which is to say $s[i-\max\dots i] = s[i-j-\max\dots i-j]$. With the max value, all of the shorter prefixes also occur too. Therefore, $dp[i] = i + 1 - \max(z[i])$. The time complexity is $O(n^2)$

```

1 def distinctSubstrs(s):
2     n = len(s)
3     if n < 1:
4         return 0
5     ans = 1 # for dp[0]
6     #last_str = s[0:1]
7     for i in range(1, n):
8         reverse_str = s[0:i+1][::-1]
9         z = linearZF(reverse_str)
10        ans += (i + 1 - max(z))
11    return ans

```

Run an example:

```

1 s = 'abab'
2 print(distinctSubstrs(s))
3 # output
4 # 7

```

22.2 Exact Multi-Patterns Matching

22.2.1 Suffix Trie/Tree/Array Introduction

Up till now, prefix function and the KMP algorithms seems impeccable with its liner time and space complexity. However, there are two problems that KMP can not resolve:

1. Approximate matching, which we will detail more in the next section.
2. If frequent queries will be made on the same text with a given pattern, and if the $m \ll n$, then KMP become impractical.

The solution to the second problem of KMP is preprocess the text and store it in order to obtain an algorithm with time complexity only related to the length of the pattern for each query. Building a suffix trie of the text is such a solution.

Suffix Trie A suffix trie of a given string is defined as:

Suffix Tree If we compress the above suffix trie, we get suffix tree.

Suffix Array Suffix Array is further applied with the benefits of saving space in storage.

Suffix Tree VS Suffix Array Each data structure has its own pros and cons. In reality, conversion between these two can be implemented in $O(n)$ time. Therefore, we can first construct one and convert it to the other later.

22.2.2 Suffix Array and Pattern Matching

Definition and Implementation

Suffix Array of a given string s is defined as all suffixes of this string in lexicographical order. Because no any two suffixes can have the same length, thus the sorting will not have equal items. For example, given $s = 'ababaa'$, the suffix array will be:

```
'a '
'aa '
'abaa '
'ababaa '
'baa '
'baba '
```

To avoid the prefix rule defined in the lexicographical order, as shown with example 'ab' < 'bab', we append a special character '\$' at the end of all suffixes. '\$' is smaller than all other characters. With this operation,

we have 'ab\$' and 'abab\$'. At position 2, '\$' will be smaller than 'a' or any other character and 'ab\$' is still smaller than 'abab\$'. Therefore, adding this special character will not lead to different sorting result, and can avoid the prefix rule when comparing two different strings.

Naive Solution with $O(n^2 \log n)$ time complexity With this knowledge, we get $s = s + '$'$, and we can generate the suffix array and sort them. A stable sorting algorithm takes $O(n \log n)$ comparison, and each comparison takes additional $O(n)$, which makes the total time complexity of $O(n^2 \log n)$.

```

1 def generateSuffixArray(s):
2     s = s + '$'
3     n = len(s)
4     suffixArray = [None]*n
5     # generate
6     for i in range(n):
7         suffixArray[i] = s[i:]
8     #print(suffixArray)
9     suffixArray.sort()
10    print(suffixArray)
11    # save space by storing the order of the suffixes , which is
12    # the starting index
13    for idx, suffix in enumerate(suffixArray):
14        suffixArray[idx] = n - len(suffix)
15    print(suffixArray)
16    return suffixArray

```

Run the above example, we will have the following output:

```
[ '$', 'a$', 'aa$', 'abaa$', 'ababaa$', 'baa$', 'babaa$' ]
[ 6, 5, 4, 2, 0, 3, 1 ]
```

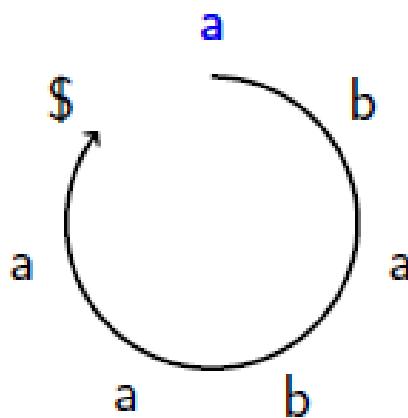


Figure 22.6: Cyclic Shifts

Cyclic Shifts For our example, we start at position 0, we get the first cyclic shift of 'ababaa\$', and then position 1, we have our second cyclic shift 'babaa\$a', and so till the last position of the string. Now, let us see what happens if we sort all of the cyclic shifts:

	Sorted	To Suffix Array
0: ababaa\$	\$ababaa	\$
1: babaa\$a	a\$ababa	a\$
2: abaa\$ab	aa\$abab	aa\$
3: baa\$aba	abaa\$ab	abaa\$
4: aa\$abab	ababaa\$	ababaa\$
5: a\$ababa	baa\$aba	baa\$
6: \$ababaa	baba@a	baba@a

We know the number of cyclic shifts is the same as of the number of all suffixes of the same string. And by observing the above example, sorting the cyclic shifts will get us sorted suffixes if we remove all characters after '\$' in each cyclic shift. This conclusion can be hold true for all strings because '\$' is smaller than all other characters, and with '\$' at different position in each cyclic shift, once we are at '\$', the comparison of two strings end because the first one that has '\$' is smaller than all others. Therefore, all the characters after the '\$' will not affect the sorting at all. Now, we know that sorting cyclic shifts and suffixes of string s is equivalent with the addition of '\$' at the end.

If we can sort the cyclic shifts of string in faster way, then we will find ourselves a more efficient suffix sorting algorithm. One obvious efficient sorting algorithm is using Radix Sort. Using radix sort, we first sort the cyclic shifts by the last character using counting sort, and then the second last character till finishing the first character. Sorting each character for the whole cyclic shifts array takes $O(n)$, and we are running n rounds, this makes the whole sorting of $O(n^2)$ and with $O(n)$ space. However, we can improve these complexity further by using special properties of the Cyclic shifts.

Partial Cyclic Shifts Different from the cyclic shifts, partial cyclic shifts are defined as C_i^L which is the partial cyclic shift of length L starting at index i . For the above example, the partial cyclic shift of length 1, 2, and 4 will be :

	C^7	C^1	C^2	C^4
ababaa\$	a	<u>a</u> <u>b</u>	<u>a</u> <u>b</u> <u>a</u>	
babaa\$a	b	<u>b</u> <u>a</u>	<u>b</u> <u>a</u> <u>b</u>	
abaa\$ab	a	<u>a</u> <u>b</u>	<u>a</u> <u>b</u> <u>a</u>	
baa\$aba	b	<u>b</u> <u>a</u>	<u>b</u> <u>a</u> <u>a</u>	
aa\$abab	a	<u>a</u> <u>a</u>	<u>a</u> <u>a</u> <u>a</u>	
a\$ababa	a	<u>a</u> <u>\$</u>	<u>a</u> <u>\$</u> <u>b</u>	
\$ababaa	\$	<u>\$</u> <u>a</u>	<u>\$</u> <u>a</u> <u>b</u>	

Carefully observing the relation of pair (C_1, C_2) and (C_2, C_4) . We can find that C_1 and the second half of substring (denoted by underline) in C_2 has the same key set. Same rule applies to C_2 and the second half of substring in C_4 .

Doubled Partial Cyclic Shifts *Doubled Partial Cyclic Shifts* of C_i^L is C_i^{2L} and $C_i^{2L} = C_i^L C_{i+L}^L$ (with concatenation of these two strings). Apply the same methodology of Radix Sort, we can sort the doubled partial shifts by firstly sort the second half and then the first half. Therefore, instead of doing n rounds of counting sort on each character, we do $\log n$ rounds of sorting of the doubled partial cyclic shifts from the last round. If we can sort each round in $O(n)$, then we make the time complexity to $O(n \log n)$ ($T(n) = T(n/2)$) which is way better than the radix sort of $O(n^2)$. The starting point of sorting doubled partial cyclic is sorting the partial cyclic shifts with length one.

Order and Class *Order* is defined as the sorted cyclic shift with the starting index as their value. For example, for C^1 , the sorted order will be $[6, 0, 2, 4, 5, 1, 3]$, which represents $[\$, a, a, a, a, b, b]$. *Class* is an array that each item $Class_i$ corresponds to C_i and denotes as the number of partial cyclic shifts of the same length that are strictly smaller than C_i . For 'ababaa\$', the class of length 1 will be $[1, 2, 1, 2, 1, 1, 0]$. The reason to bring in the concept of class is because of the rule that the set of first and second half of the doubled partial cyclic shifts share the same key set, and **the class is equivalent to the converted key of corresponding partial cyclic shift**.

Compute Order and Class of Partial Cyclic Shifts of Length 1
For C^1 , we can obtain with counting sort with the range of 256 for all common English characters. For C^1 , we know for $[\$, a, a, a, a, b, b]$, we assign order as $[0, 1, 1, 1, 1, 2, 2]$. Each one corresponds to C_{order_i} . Because the class corresponds to the original string order, therefore, we just need to put these class back to $order_i$ position in the array. We recap this as: we first set $class[order[0]] = 0$, and looping over the order array from $[1, n-1]$, the corresponding character will be $s[order[i]]$ and the last order char will be $s[order[i-1]]$. We just need to compare if it equals.

```
if s[order[i]] != s[order[i-1]]:
    # use order as index to put the result back
    class[order[i]] = class[order[i-1]] + 1
else:
    class[order[i]] = class[order[i-1]]
```

The Python implementation of Computing Order for Partial Cyclic Shift of Length 1, the time complexity is $O(n + k)$, k is the number of possible characters.

```

1 def getCharOrder(s):
2     n = len(s)
3     numChars = 256
4     count = [0]*numChars # totally 256 chars , if you want , can
5         print it out to see these chars
6
7     order = [0]*(n)
8
9     #count the occurrence of each char
10    for c in s:
11        count[ord(c)] += 1
12
13    # prefix sum of each char
14    for i in range(1, numChars):
15        count[i] += count[i-1]
16
17    # assign from count down to be stable
18    for i in range(n-1,-1,-1):
19        count[ord(s[i])] -=1
20        order[count[ord(s[i])]] = i # put the index into the order
21        instead the suffix string
22
23    return order

```

The Python implementation of Computing Class for Partial Cyclic Shift of Length 1, this can be applied in $O(n)$ given the order.

```

1 def getCharClass(s, order):
2     n = len(s)
3     cls = [0]*n
4     # if it all differs , then cls[i] = order[i]
5     cls[order[0]] = 0 #the 6th will be 0
6     for i in range(1, n):
7         # use order[i] as index , so the last index
8         if s[order[i]] != s[order[i-1]]:
9             print('diff',s[order[i]],s[order[i-1]])
10            cls[order[i]] = cls[order[i-1]] + 1
11        else:
12            cls[order[i]] = cls[order[i-1]]
13    return cls

```

Applying the above two functions, we can get:

L=1	cls	order	CL=2	order	cls
i=0, a:1	\$:6		a\$:5	\$a:6	ab:3
i=1, b:2	a:0		\$a:6	a\$:5	ba:4
i=2, a:1	a:2		ba:1	aa:4	ab:3
i=3, b:2	a:4		ba:3	ab:0	ba:4
i=4, a:1	a:5		aa:4	ab:2	aa:2
i=5, a:1	b:1		ab:0	ba:1	a\$:1
i=6, \$:0	b:3		ab:2	ba:3	\$a:0

Sort the Doubled Partial Cyclic shifts To apply radix sorting, we double our previous sorted partial shifts of C_i^L as $C_{i-L}^L C_i^L$. Given the fact

that the second part C_i^L is already sorted, we just need to sort the first half with counting sort using the class array of the last partial cyclic shifts. The time complexity of this step is $O(n)$ too. The Python implementation of computing the doubled partial cyclic shifts' order is:

```

1 '''It is a counting sort using the first part as class'''
2 def sortDoubled(s, L, order, cls):
3     n = len(s)
4     count = [0] * n
5     new_order = [0] * n
6     # their key is the class
7     for i in range(n):
8         count[cls[i]] += 1
9
10    # prefix sum
11    for i in range(1, n):
12        count[i] += count[i-1]
13
14    # assign from count down to be stable
15    # sort the first half
16    for i in range(n-1, -1, -1):
17        start = (order[i] - L + n) % n #get the start index of the
18        # first half,
19        count[cls[start]] -= 1
20        new_order[count[cls[start]]] = start
21
22    return new_order

```

Now, similarly, we compute the new class information. The comparison of the string is converted to compare its corresponding class info, as a pair (P_1, P_2) which is the class of the first and second half.

```

1 def updateClass(order, cls, L):
2     n = len(order)
3     new_cls = [0]*n
4     # if it all differs, then cls[i] = order[i]
5     new_cls[order[0]] = 0 #the 6th will be 0
6     for i in range(1, n):
7         cur_order, prev_order = order[i], order[i-1]
8         # use order[i] as index, so the last index
9         if cls[cur_order] != cls[prev_order] or cls[(cur_order+L) %
10             n] != cls[(prev_order+L) % n]:
11             new_cls[cur_order] = new_cls[prev_order] + 1
12         else:
13             new_cls[cur_order] = new_cls[prev_order]
14
15     return new_cls

```

Sorting Cyclic Shifts in $O(n \log n)$ Now, we have derived ourselves a $O(n \log n)$ suffix array construction algorithm. We start from sorting partial cyclic shifts of length 1 and each time to double the length until the the sorted length is \geq to the string's length.

```

1 def cyclic_shifts_sort(s):

```

```

2   s = s + '$'
3   n = len(s)
4   order = getCharOrder(s)
5   cls = getCharClass(s, order)
6   print(order, cls)
7   L = 1
8   while L < n:
9       order = sortDoubled(s, 1, order, cls)
10      cls = updateClass(order, cls, L)
11      print(order, cls)
12      L *= 2
13
14  return order

```

Applications

Number of Distinct Substrings of a string

22.2.3 Rabin-Karp Algorithm (Exact or anagram Pattern Matching)

Used to find the exact pattern, because different anagram of string would have different hash value.

22.3 Bonus

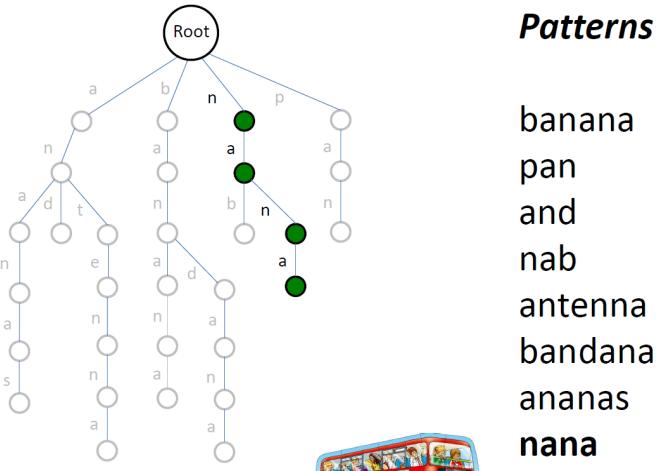


Figure 22.7: Building a Trie from Patterns

Multiple-Patterns Matching Previously, we mainly talked about exact/approximate one pattern matching. When there are multiple patterns the

time complexity became to $O(\sum_i m_i * n)$ if brute force solution is used. We can construct a trie of all patterns as shown in Section 27.3. For example, in Fig. 22.7 shows a trie built with all patterns.

Now, let us do **Trie Matching** exactly the same way as the brute force pattern matching algorithm by sliding the pattern trie along the text at each position of text. Each comparison: walk down the trie by spelling symbols of text and a pattern from the pattern list matches text each time we reach a leaf. Try text = “panamabanananas”. We will first walk down branch of p->a->n and stop at the leaf, thus we find pattern ‘pan’. With Trie Matching, the runtime is decreased to $O(\max_i m_i * n)$. Plus the trie construction time $O(\sum_i m_i)$.

However, merging all patterns into a trie makes it impossible for using advanced single-pattern matching algorithms such as KMP.

More Pattern Matching Tasks There are more types of matching, instead of finding the exact occurrence of one string in another.

1. Longest Common Substring (LCS): LCS asks us to return the longest substring between these two strings.
2. Anagram Matching: this asks us to find a substring in T that has all letters in P, and does not care about the order of these letters in P.
3. Palindrome Matching.

22.4 Trie for String

Definition Trie comes from the word reTrieval. In computer science, a trie, also called digital tree, radix tree or prefix tree which like BST is also a kind of search tree for finding substring in a text. We can solve string matching in $O(|T|)$ time, where $|T|$ is the size of our text. This purely algorithmic approach has been studied extensively in the algorithms: Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp. However, we entertain the possibility that multiple queries will be made to the same text. This motivates the development of data structures that preprocess the text to allow for more efficient queries. Such efficient data structure is Trie, which can do each query in $O(P)$, where P is the length of the pattern string. Trie is an ordered tree structure, which is used mostly for storing strings (like words in dictionary) in a compact way.

1. In a Trie, each child branch is labeled with letters in the alphabet Σ . Actually, it is not necessary to store the letter as the key, because if we order the child branches of every node alphabetically from left to right, the position in the tree defines the key which it is associated to.

2. The root node in a Trie represents an empty string.

Now, we define a trie Node: first it would have a bool variable to denote if it is the end of the word and a children which is a list of 26 children TrieNodes.

```

1 class TrieNode:
2     # Trie node class
3     def __init__(self):
4         self.children = [None]*26
5         # isEndOfWord is True if node represent the end of the
6         # word
6         self.isEndOfWord = False

```

Compressed Trie

- Compress unary nodes, label edges by strings

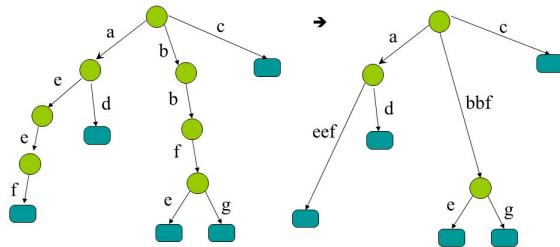


Figure 22.8: Trie VS Compact Trie

Compact Trie If we assign only one letter per edge, we are not taking full advantage of the trie's tree structure. It is more useful to consider compact or compressed tries, tries where we remove the one letter per edge constraint, and contract non-branching paths by concatenating the letters on these paths. In this way, every node branches out, and every node traversed represents a choice between two different words. The compressed trie that corresponds to our example trie is also shown in Figure 27.4.

Operations: INSERT, SEARCH Both for INSERT and SEARCH, it takes $O(m)$, where m is the length of the word/string we want to insert or search in the trie. Here, we use an LeetCode problem as an example showing how to implement INSERT and SEARCH. Because constructing a trie is a series of INSERT operations which will take $O(n * m)$, n is the total numbers of words/strings, and m is the average length of each item. The space complexity for the non-compact Trie would be $O(N * |\Sigma|)$, where $|\Sigma|$ is the alphabetical size, and N is the total number of nodes in the trie structure. The upper bound of N is $n * m$.

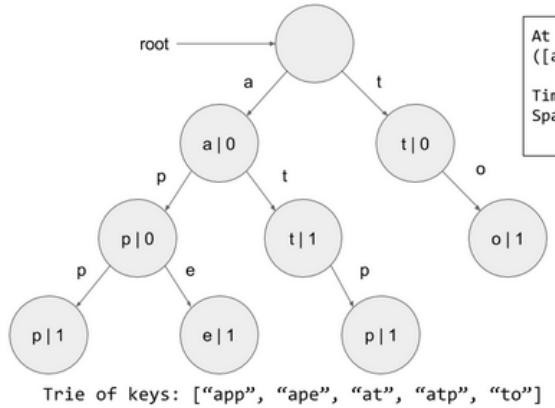


Figure 22.9: Trie Structure

22.1 208. Implement Trie (Prefix Tree) (medium). Implement a trie with insert, search, and startsWith methods.

```

1 Example:
2 Trie trie = new Trie();
3 trie.insert("apple");
4 trie.search("apple");    // returns true
5 trie.search("app");     // returns false
6 trie.startsWith("app"); // returns true
7 trie.insert("app");
8 trie.search("app");     // returns true

```

Note: You may assume that all inputs are consist of lowercase letters a-z. All inputs are guaranteed to be non-empty strings.

INSERT with INSERT operation, we woould be able to insert a given word in the trie, when traversing the trie from the root node which is a `TrieNode`, with each letter in word, if its corresponding node is `None`, we need to put a node, and continue. At the end, we need to set that node's `endofWord` variable to `True`. thereafter, we would have a new branch starts from that node constructed. For example, when we first insert “app“ as shown in Fig 27.4, we would end up building branch “app“, and with ape, we would add nodes “e“ as demonstrated with red arrows.

```

1 def insert(self, word):
2     """
3     Inserts a word into the trie.
4     :type word: str
5     :rtype: void
6     """
7     node = self.root #start from the root node
8     for c in word:

```

```
9     loc = ord(c)-ord('a')
10    if node.children[loc] is None: # char does not
11        exist, new one
12        node.children[loc] = self.TrieNode()
13    # move to the next node
14    node = node.children[loc]
15    # set the flag to true
16    node.is_word = True
```

SEARCH For SEARCH, like INSERT, we traverse the trie using the letters as pointers to the next branch. There are three cases: 1) for word P, if it doesn't exist, but its prefix does exist, then we return False. 2) If we found a matching for all the letters of P, at the last node, we need to check if it is a leaf node where `is_word` is True. STARTWITH is just slightly different from SEARCH, it does not need to check that and return True after all letters matched.

```
def search(self, word):
    node = self.root
    for c in word:
        loc = ord(c)-ord('a')
        # case 1: not all letters matched
        if node.children[loc] is None:
            return False
        node = node.children[loc]
    # case 2
    return True if node.is_word else False
```

```
1 def startWith(self, word):
2     node = self.root
3     for c in word:
4         loc = ord(c)-ord('a')
5         # case 1: not all letters matched
6         if node.children[loc] is None:
7             return False
8         node = node.children[loc]
9     # case 2
10    return True
```

Now complete the given Trie class with TrieNode and `__init__` function.

```
1 class Trie:
2     class TrieNode:
3         def __init__(self):
4             self.is_word = False
5             self.children = [None] * 26 #the order of the
node represents a char
6
7     def __init__(self):
8         """
9             Initialize your data structure here.

```

```

10     """
11     self.root = self.TrieNode() # root has value None

```

22.1 336. Palindrome Pairs (hard). Given a list of unique words, find all pairs of distinct indices (i, j) in the given list, so that the concatenation of the two words, i.e. $\text{words}[i] + \text{words}[j]$ is a palindrome.

```

1 Example 1:
2
3 Input: ["abcd", "dcba", "lls", "s", "sssll"]
4 Output: [[0,1],[1,0],[3,2],[2,4]]
5 Explanation: The palindromes are ["dcbaabcd", "abcddcba",
   "slls", "llssssll"]
6
7 Example 2:
8
9 Input: ["bat", "tab", "cat"]
10 Output: [[0,1],[1,0]]
11 Explanation: The palindromes are ["battab", "tabbat"]

```

Solution: One Forward Trie and Another Backward Trie. We start from the naive solution, which means for each element, we check if it is palindrome with all the other strings. And from the example 1, [3,3] can be a pair, but it is not one of the outputs, which means this is a combination problem, the time complexity is $C_n C_{n-1}$, and multiply it with the average length of all the strings, we make it m , which makes the complexity to be $O(mn^2)$. However, we can use Trie Structure,

```

1 from collections import defaultdict
2
3
4 class Trie:
5     def __init__(self):
6         self.links = defaultdict(self.__class__)
7         self.index = None
8         # holds indices which contain this prefix and whose
9         # remainder is a palindrome
10        self.pali_indices = set()
11
12    def insert(self, word, i):
13        trie = self
14        for j, ch in enumerate(word):
15            trie = trie.links[ch]
16            if word[j+1:] and is_palindrome(word[j+1:]):
17                trie.pali_indices.add(i)
18        trie.index = i
19
20    def is_palindrome(word):
21        i, j = 0, len(word) - 1
22        while i <= j:

```

```

23         if word[i] != word[j]:
24             return False
25         i += 1
26         j -= 1
27     return True
28
29
30 class Solution:
31     def palindromePairs(self, words):
32         '''Find pairs of palindromes in O(n*k^2) time and O
33         (n*k) space.'''
34         root = Trie()
35         res = []
36         for i, word in enumerate(words):
37             if not word:
38                 continue
39             root.insert(word[::-1], i)
40         for i, word in enumerate(words):
41             if not word:
42                 continue
43             trie = root
44             for j, ch in enumerate(word):
45                 if ch not in trie.links:
46                     break
47                 trie = trie.links[ch]
48                 if is_palindrome(word[j+1:]) and trie.index
49                 is not None and trie.index != i:
50                     # if this word completes to a
51                     # palindrome and the prefix is a word, complete it
52                     res.append([i, trie.index])
53             else:
54                 # this word is a reverse suffix of other
55                 # words, combine with those that complete to a palindrome
56                 for pali_index in trie.pali_indices:
57                     if i != pali_index:
58                         res.append([i, pali_index])
59         if '' in words:
60             j = words.index('')
61             for i, word in enumerate(words):
62                 if i != j and is_palindrome(word):
63                     res.append([i, j])
64                     res.append([j, i])
65     return res

```

Solution2: Moreover, there are always more clever ways to solve these problems. Let us look at a clever way: abcd, the prefix is ". 'a', 'ab', 'abc', 'abcd', if the prefix is a palindrome, so the reverse[abcd], reverse[dc], to find them in the words, the words stored in the words with index is fastest to find. $O(n)$. Note that when considering suffixes, we explicitly leave out the empty string to avoid counting duplicates. That is, if a palindrome can be created by appending an entire other word to the current word, then we will already consider such a

palindrome when considering the empty string as prefix for the other word.

```

1  class Solution(object):
2      def palindromePairs(self, words):
3          # 0 means the word is not reversed, 1 means the
4          # word is reversed
5          words, length, result = sorted([(w, 0, i, len(w))
6                                           for i, w in enumerate(words)] +
7                                           [(w[::-1], 1, i, len(w))
8                                           for i, w in enumerate(words)]), len(words) * 2, []
9
10         #after the sorting ,the same string were nearby , one
11         #is 0 and one is 1
12         for i, (word1, rev1, ind1, len1) in enumerate(words
13             ):
14             for j in xrange(i + 1, length):
15                 word2, rev2, ind2, _ = words[j]
16                 #print word1, word2
17                 if word2.startswith(word1): # word2 might
18                     be longer
19                     if ind1 != ind2 and rev1 ^ rev2: # one
20                         is reversed one is not
21                         rest = word2[len1:]
22                         if rest == rest[::-1]: result += ([
23                             ind1, ind2],)
24                         if rev2 else ([ind2, ind1],) # if rev2 is
25                         reversed , the from ind1 to ind2
26                         else:
27                             break # from the point of view , break
28                         is powerful , this way , we only deal with possible
29                         reversed ,
30             return result
31
32

```

There are several other data structures, like balanced trees and hash tables, which give us the possibility to search for a word in a dataset of strings. Then why do we need trie? Although hash table has $O(1)$ time complexity for looking for a key, it is not efficient in the following operations :

- Finding all keys with a common prefix.
- Enumerating a dataset of strings in lexicographical order.

Sorting Lexicographic sorting of a set of keys can be accomplished by building a trie from them, and traversing it in pre-order, printing only the leaves' values. This algorithm is a form of radix sort. This is why it is also called radix tree.

Part VI

Math and Geometry

23

Math and Probability Problems

In this chapter, we will specifically talk math related problems. Normally, for the problems appearing in this section, they can be solved using our learned programming methodology. However, it might not be efficient (we will get LTE error on the LeetCode) due to the fact that we are ignoring their math properties which might help us boost the efficiency. Thus, learning some of the most related math knowledge can make our life easier.

23.1 Numbers

23.1.1 Prime Numbers

A prime number is an integer greater than 1, which is only divisible by 1 and itself. First few prime numbers are : 2 3 5 7 11 13 17 19 23 ...

Some interesting facts about Prime numbers:

1. 2 is the only even Prime number.
2. 2, 3 are only two consecutive natural numbers which are prime too.
3. Every prime number except 2 and 3 can be represented in form of $6n+1$ or $6n-1$, where n is natural number.
4. Goldbach Conjecture: Every even integer greater than 2 can be expressed as the sum of two primes. Every positive integer can be decomposed into a product of primes.
5. GCD of a natural number with Prime is always one.

6. Fermat's Little Theorem: If n is a prime number, then for every a , $1 \leq a < n$,
7. Prime Number Theorem : The probability that a given, randomly chosen number n is prime is inversely proportional to its number of digits, or to the logarithm of n .

Check Single Prime Number

Learning to check if a number is a prime number is necessary: the naive solution comes from the direct definition, for a number n , we try to check if it can be divided by number in range $[2, n - 1]$, if it divides, then its not a prime number.

```

1 def isPrime(n):
2     # Corner case
3     if (n <= 1):
4         return False
5     # Check from 2 to n-1
6     for i in range(2, n):
7         if (n % i == 0):
8             return False
9     return True

```

There are actually a lot of space for us to optimize the algorithm. First, instead of checking till n , we can check till \sqrt{n} because a larger factor of n must be a multiple of smaller factor that has been already checked. Also, because even numbers bigger than 2 are not prime, so the step we can set it to 2. The algorithm can be improved further by use feature 3 that all primes are of the form $6k \pm 1$, with the exception of 2 and 3. Together with feature 4 which implicitly states that every non-prime integer is divisible by a prime number smaller than itself. So a more efficient method is to test if n is divisible by 2 or 3, then to check through all the numbers of form $6k \pm 1$.

```

1 def isPrime(n):
2     # corner cases
3     if n <= 1:
4         return False
5     if n<= 3:
6         return True
7
8     if n % 2 == 0 or n % 3 == 0:
9         return False
10
11    for i in range(5, int(n**0.5)+1, 6): # 6k+1 or 6k-1, step
12        if n%i == 0 or n%(i+2)==0:
13            return False
14    return True
15 return True

```

Generate A Range of Prime Numbers

Wilson theorem says if a number k is prime then $((k - 1)! + 1) \% k$ must be 0. Below is Python implementation of the approach. Note that the solution works in Python because Python supports large integers by default therefore factorial of large numbers can be computed.

```

1 # Wilson Theorem
2 def primesInRange(n):
3     fact = 1
4     rst = []
5     for k in range(2, n):
6         fact *= (k-1)
7         if (fact + 1)% k == 0:
8             rst.append(k)
9     return rst
10
11 print(primesInRange(15))
12 # output
13 # [2, 3, 5, 7, 11, 13]
```

Sieve Of Eratosthenes To generate a list of primes. It works by recognizing *Goldbach Conjecture* that all non-prime numbers are divisible by a prime number. An optimization is to only use odd number in the primes list, so that we can save half space and half time. The only difference is we need to do index mapping.

```

1 def primesInRange(n):
2     primes = [True] * n
3     primes[0] = primes[1] = False
4     for i in range(2, int(n ** 0.5) + 1):
5         #cross off remaining multiples of prime i, start with i*i
6         if primes[i]:
7             for j in range(i*i, n, i):
8                 primes[j] = False
9     rst = [] # or use sum(primes) to get the total number
10    for i, p in enumerate(primes):
11        if p:
12            rst.append(i)
13    return rst
14
15 print(primesInRange(15))
```

23.1.2 Ugly Numbers

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. We can write it as $\text{ugly number} = 2^i 3^j 5^k$, $i \geq 0, j \geq 0, k \geq 0$. Examples of ugly numbers: 1, 2, 3, 5, 6, 10, 15, ... The concept of ugly number is quite simple. Now let us use the LeetCode problems as example to derive the algorithms to identify ugly numbers.

Check a Single Number

263. Ugly Number (Easy)

```

1 Ugly numbers are positive numbers whose prime factors only
   include 2, 3, 5. For example, 6, 8 are ugly while 14 is not
   ugly since it includes another prime factor 7.
2
3 Note:
4     1 is typically treated as an ugly number.
5     Input is within the 32-bit signed integer range.

```

Analysis: because the ugly number is only divisible by 2,3,5, so if we keep dividing the number by these factors (num/f), eventually we would get 1, if the remainder ($num\%f$) is 0 (divisible), otherwise we stop the loop to check the number.

```

1 def isUgly(self, num):
2     """
3         :type num: int
4         :rtype: bool
5     """
6     if num == 0:
7         return False
8     factor = [2, 3, 5]
9     for f in factor:
10        while num % f == 0:
11            num /= f
12    return num == 1

```

Generate A Range of Number

264. Ugly Number II (medium)

```

1 Write a program to find the n-th ugly number.
2
3 Ugly numbers are positive numbers whose prime factors only
   include 2, 3, 5. For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12
   is the sequence of the first 10 ugly numbers.
4
5 Note that 1 is typically treated as an ugly number, and n does
   not exceed 1690.

```

Analysis: The first solution is we use the rules $uglynumber = 2^i3^j5^k, i \geq 0, j \geq 0, k \geq 0$, using three for loops to generate at least 1690 ugly numbers that is in the range of 2^{32} , and then sort them, the time complexity is $O(n \log n)$, with $O(n)$ in space. However, if we need to constantly make request, it seems reasonable to save a table, and once the table is generated and saved, each time we would only need constant time to check.

```

1 from math import log, ceil
2 class Solution:
3     ugly = [2**i * 3**j * 5**k for i in range(32) for j in range(
4         (ceil(log(2**32, 3))) for k in range(ceil(log(2**32, 5))))]

```

```

4     ugly.sort()
5     def nthUglyNumber(self, n):
6         """
7             :type n: int
8             :rtype: int
9         """
10    return self.ugly[n-1]

```

The second way is only generate the nth ugly number, with

```

1 class Solution:
2     n = 1690
3     ugly = [1]
4     i2 = i3 = i5 = 0
5     for i in range(n-1):
6         u2, u3, u5 = 2 * ugly[i2], 3 * ugly[i3], 5 * ugly[i5]
7         umin = min(u2, u3, u5)
8         ugly.append(umin)
9         if umin == u2:
10             i2 += 1
11         if umin == u3:
12             i3 += 1
13         if umin == u5:
14             i5 += 1
15
16     def nthUglyNumber(self, n):
17         """
18             :type n: int
19             :rtype: int
20         """
21     return self.ugly[n-1]

```

23.1.3 Combinatorics

1. 611. Valid Triangle Number

23.1 Pascal's Triangle II(L119, *). Given a non-negative index k where $k \leq 33$, return the k th index row of the Pascal's triangle. Note that the row index starts from 0. In Pascal's triangle, each number is the sum of the two numbers directly above it.

Example:

Input: 3

Output: [1, 3, 3, 1]

Follow up: Could you optimize your algorithm to use only $O(k)$ extra space? **Solution: Generate from Index 0 to K.**

```

1 def getRow(self, rowIndex):
2     if rowIndex == 0:
3         return [1]
4     # first, n = rowIndex+1, if n is even,
5     ans = [1]

```

```

6     for i in range(rowIndex):
7         tmp = [1]*(i+2)
8         for j in range(1, i+1):
9             tmp[j] = ans[j-1]+ans[j]
10        ans = tmp
11    return ans

```

Triangle Counting

Smallest Larger Number

556. Next Greater Element III

```

1 Given a positive 32-bit integer n, you need to find the smallest
2 32-bit integer which has exactly the same digits existing in
3 the integer n and is greater in value than n. If no such
4 positive 32-bit integer exists, you need to return -1.
5
6 Example 1:
7
8 Input: 12
9 Output: 21
10
11 Example 2:
12
13 Input: 21
14 Output: -1

```

Analysis: The first solution is to get all digits [1,2], and generate all the permutation [[1,2],[2,1]], and generate the integer again, and then sort generated integers, so that we can pick the next one that is larger. But the time complexity is $O(n!)$.

Now, let us think about more examples to find the rule here:

```

1 435798->435879
2 1432->2134

```

If we start from the last digit, we look to its left, find the closest digit that has smaller value, we then switch this digit, if we can't find such digit, then we search the second last digit. If none is found, then we can not find one. Like 21. return -1. This process is we get the first larger number to the right.

```

1 [5, 5, 7, 8, -1, -1]
2 [2, -1, -1, -1]

```

After this we switch 8 with 7: we get

```

1 4358 97
2 2 431

```

For the remaining digits, we do a sorting and put them back to those digit to get the smallest value

```

1 class Solution:
2     def getDigits(self, n):
3         digits = []
4         while n:
5             digits.append(n%10) # the least important position
6             n = int(n/10)
7         return digits
8     def getSmallestLargerElement(self, nums):
9         if not nums:
10             return []
11         rst = [-1]*len(nums)
12
13     for i, v in enumerate(nums):
14         smallestLargerNum = sys.maxsize
15         index = -1
16         for j in range(i+1, len(nums)):
17             if nums[j]>v and smallestLargerNum > nums[j]:
18                 index = j
19                 smallestLargerNum = nums[j]
20         if smallestLargerNum < sys.maxsize:
21             rst[i] = index
22     return rst
23
24
25     def nextGreaterElement(self, n):
26         """
27         :type n: int
28         :rtype: int
29         """
30         if n==0:
31             return -1
32
33         digits = self.getDigits(n)
34         digits = digits[::-1]
35         # print(digits)
36
37         rst = self.getSmallestLargerElement(digits)
38         # print(rst)
39         stop_index = -1
40
41         # switch
42         for i in range(len(rst)-1, -1, -1):
43             if rst[i]!=-1: #switch
44                 print('switch')
45                 stop_index = i
46                 digits[i], digits[rst[i]] = digits[rst[i]], digits[i]
47                 break
48         if stop_index == -1:
49             return -1
50
51         # print(digits)
52
53         # sort from stop_index+1 to the end

```

```

54     digits[stop_index+1:] = sorted(digits[stop_index+1:])
55     print(digits)
56
57 #convert the digitalized answer to integer
58     nums = 0
59     digit = 1
60     for i in digits[::-1]:
61         nums+=digit*i
62         digit*=10
63         if nums>2147483647:
64             return -1
65
66
67     return nums

```

23.2 Intersection of Numbers

In this section, intersection of numbers is to find the “common” thing between them, for example Greatest Common Divisor and Lowest Common Multiple.

23.2.1 Greatest Common Divisor

GCD (Greatest Common Divisor) or HCF (Highest Common Factor) of two numbers a and b is the largest number that divides both of them. For example shown as follows:

- 1 The divisors of 36 are: 1, 2, 3, 4, 6, 9, 12, 18, 36
- 2 The divisors of 60 are: 1, 2, 3, 4, 5, 6, 10, 12, 15, 30, 60
- 3 GCD = 12

Special case is when one number is zero, the GCD is the value of the other.
 $gcd(a, 0) = a$.

The basic algorithm is: we get all divisors of each number, and then find the largest common value. Now, let’s see how to we advance this algorithm. We can reformulate the last example as:

- 1 $36 = 2 * 2 * 3 * 3$
- 2 $60 = 2 * 2 * 3 * 5$
- 3 $GCD = 2 * 2 * 3$
- 4 $= 12$

So if we use $60 - 36 = 2*2*3*5 - 2*2*3*3 = (2*2*3)*(5-3) = 2*2*3*2$. So we can derive the principle that the GCD of two numbers does not change if the larger number is replaced by its difference with the smaller number. The features of GCD:

1. $gcd(a, 0) = a$
2. $gcd(a, a) = a$,

3. $\gcd(a, b) = \gcd(a - b, b)$, if $a > b$.

Based on the above features, we can use Euclidean Algorithm to gain GCD:

```

1 def euclid(a, b):
2     while a != b:
3         # replace larger number by its difference with the
4         # smaller number
5         if a > b:
6             a = a - b
7         else:
8             b = b - a
9     return a
10 print(euclid(36, 60))

```

The only problem with the Euclidean Algorithm is that it can take several subtraction steps to find the GCD if one of the given numbers is much bigger than the other. A more efficient algorithm is to replace the subtraction with remainder operation. The algorithm would stop when reaching a zero remainder and now the algorithm never requires more steps than five times the number of digits (base 10) of the smaller integer.

The recursive version code:

```

1 def euclidRemainder(a, b):
2     if a == 0 :
3         return b
4     return gcd(b%a, a)

```

The iterative version code:

```

1 def euclidRemainder(a, b):
2     while a > 0:
3         # replace one number with remainder between them
4         a, b = b%a, a
5     return b
6
7 print(euclidRemainder(36, 60))

```

23.2.2 Lowest Common Multiple

Lowest Common Multiple (LCM) is the smallest number that is a multiple of both a and b . For example of 6 and 8:

```

1 The multiplies of 6 are: 6, 12, 18, 24, 30, ...
2 The multiplies of 8 are: 8, 16, 24, 32, 40, ...
3 LCM = 24

```

Computing LCM is dependent on the GCD with the following formula:

$$\text{lcm}(a, b) = \frac{a \times b}{\gcd(a, b)} \quad (23.1)$$

23.3 Arithmetic Operations

Because for the computer, it only understands the binary representation as we learned in Bit Manipulation (Chapter III, the most basic arithmetic operation it supports are binary addition and subtraction. (Of course, it can execute the bit manipulation too.) The other common arithmetic operations such as Multiplication, division, modulus, exponent are all implemented/-coded with the addition and subtraction as basis or in a dominant fashion. As a software engineer, have a sense of how we can implement the other operations from the given basis is reasonable and a good practice of the coding skills. Also, sometimes if the factor to compute on is extra large number, which is to say the computer can not represent, we can still compute the result by treating these numbers as strings.

In this section, we will explore operations include multiplication, division. There are different algorithms that we can use, we learn a standard one called long multiplication and long division. I am assuming you know the algorithms and focusing on the implementation of the code instead.

Long Multiplication

Long Division We treat the dividend as a string, e.g. dividend = 3456, and the divisor = 12. We start with 34, which has the digits as of divisor. $34/12 = 2, 10$, where 2 is the integer part and 10 is the reminder. Next step, we take the reminder and join with the next digit in the dividend, we get $105/12 = 8, 9$. Similarly, $96/12 = 8, 0$. Therefore we get the results by joining the result of each dividend operation, '288'. To see the coding, let us code it the way required by the following LeetCode Problem. In the process we need ($n-m$) (n, m is the total number of digits of dividend and divisor, respectively) division operation. Each division operation will be done at most 9 steps. This makes the time complexity $O(n - m)$.

23.2 29. Divide Two Integers (medium) Given two integers dividend and divisor, divide two integers without using multiplication, division and mod operator. Return the quotient after dividing dividend by divisor. The integer division should truncate toward zero.

```

1 Example 1:
2
3 Input: dividend = 10, divisor = 3
4 Output: 3
5
6 Example 2:
7
8 Input: dividend = 7, divisor = -3
9 Output: -2

```

Analysis: we can get the sign of the result first, and then convert the dividend and divisor into its absolute value. Also, we better handle the bound condition that the divisor is larger than the dividend, we get 0 directly. The code is given:

```

1 def divide(self, dividend, divisor):
2     def divide(dd): # the last position that divisor* val <
3         dd
4         s, r = 0, 0
5         for i in range(9):
6             tmp = s + divisor
7             if tmp <= dd:
8                 s = tmp
9             else:
10                return str(i), str(dd-s)
11        return str(9), str(dd-s)
12
13    if dividend == 0:
14        return 0
15    sign = -1
16    if (dividend >0 and divisor >0 ) or (dividend < 0 and
17 divisor < 0):
18        sign = 1
19    dividend = abs(dividend)
20    divisor = abs(divisor)
21    if divisor > dividend:
22        return 0
23    ans, did, dr = [], str(dividend), str(divisor)
24    n = len(dr)
25    pre = did[:n-1]
26    for i in range(n-1, len(did)):
27        dd = pre+did[i]
28        dd = int(dd)
29        v, pre = divide(dd)
30        ans.append(v)
31
32    ans = int(''.join(ans))*sign
33
34    if ans > (1<<31)-1:
35        ans = (1<<31)-1
36    return ans

```

23.4 Probability Theory

In programming tasks, such problems are either solvable with some closed-form formula or one has no choice than to enumerate the complete search space.

23.5 Linear Algebra

Gaussian Elimination is one of the several ways to find the solution for a system of linear equations.

23.6 Geometry

In this section, we will discuss coordinate related problems.

939. Minimum Area Rectangle(Medium)

Given a set of points in the xy-plane, determine the minimum area of a rectangle formed from these points, with sides parallel to the x and y axes.

If there isn't any rectangle, return 0.

```
1 Example 1:  
2  
3 Input: [[1,1],[1,3],[3,1],[3,3],[2,2]]  
4 Output: 4  
5  
6 Example 2:  
7  
8 Input: [[1,1],[1,3],[3,1],[3,3],[4,1],[4,3]]  
9 Output: 2
```

Combination. This at first it is a combination problem, we pick four points and check if it is a rectangle and then what is the size. However the time complexity can be C_n^k , which will be $O(n^4)$. The following code implements the best combination we get, however, we receive LTE:

```
1 def minAreaRect(self, points):
2     def combine(points, idx, curr, ans): # h and w at first is
3         -1
4         if len(curr) >= 2:
5             lx, rx = min([x for x, _ in curr]), max([x for x, _ in curr])
6             ly, hy = min([y for _, y in curr]), max([y for _, y in curr])
7             size = (rx-lx)*(hy-ly)
8             if size >= ans[0]:
9                 return
10            xs = [lx, rx]
11            ys = [ly, hy]
12            for x, y in curr:
13                if x not in xs or y not in ys:
14                    return
15
16            if len(curr) == 4:
17                ans[0] = min(ans[0], size)
18                return
19
20        for i in range(idx, len(points)):
21            if len(curr) <= 3:
```

```

21         combine(points, i+1, curr+[points[i]], ans)
22     return
23
24     ans=[sys.maxsize]
25     combine(points, 0, [], ans)
26     return ans[0] if ans[0] != sys.maxsize else 0

```

Math: Diagonal decides a rectangle. We use the fact that if we know the two diagonal points, say $(1, 2), (3, 4)$. Then we need $(1, 4), (3, 2)$ to make it a rectangle. If we save the points in a hashmap, then the time complexity can be decreased to $O(n^2)$. The condition that two points are diagonal is: $x_1 \neq x_2, y_1 \neq y_2$. If one of them is equal, then they form a vertical or horizontal line. If both equal, then its the same points.

```

1 class Solution(object):
2     def minAreaRect(self, points):
3         S = set(map(tuple, points))
4         ans = float('inf')
5         for j, p2 in enumerate(points): # decide the second
6             point
7                 for i in range(j): # decide the first point
8                     p1 = points[i]
9                     if (p1[0] != p2[0] and p1[1] != p2[1] and #
10                         avoid
11                         (p1[0], p2[1]) in S and (p2[0], p1[1])
12                         in S):
13                             ans = min(ans, abs(p2[0] - p1[0]) * abs(p2
14                             [1] - p1[1]))
15             return ans if ans < float('inf') else 0

```

Math: Sort by column. Group the points by x coordinates, so that we have columns of points. Then, for every pair of points in a column (with coordinates (x,y_1) and (x,y_2)), check for the smallest rectangle with this pair of points as the rightmost edge. We can do this by keeping memory of what pairs of points we've seen before.

```

1 def minAreaRect(self, points):
2     columns = collections.defaultdict(list)
3     for x, y in points:
4         columns[x].append(y)
5     lastx = {} # one-pass hash
6     ans = float('inf')
7
8     for x in sorted(columns): # sort by the keys
9         column = columns[x]
10        column.sort() # sort column
11        for j, y2 in enumerate(column): # right most edge, up
12            point
13                for i in xrange(j): # right most edge, lower
14                    point
15                        y1 = column[i]
16                        if (y1, y2) in lastx: # 1: [1, 3], will be
17                            saved, when we were at 3: [1, 3], we can get the answer

```

```

15             ans = min(ans, (x - lastx[y1, y2]) * (y2 - y1)
16             lastx[y1, y2] = x # y1, y2 form a tuple
17     return ans if ans < float('inf') else 0

```

23.7 Miscellaneous Categories

23.7.1 Floyd's Cycle-Finding Algorithm

Without this we detect cycle with the following code:

```

1 def detectCycle(self, A):
2     visited=set()
3     head=point=A
4     while point:
5         if point.val in visited:
6             return point
7         visited.add(point)
8         point=point.next
9     return None

```

Traverse linked list using two pointers. Move one pointer by one and other pointer by two. If these pointers meet at some node then there is a loop. If pointers do not meet then linked list doesn't have loop. Once you detect a cycle, think about finding the starting point.

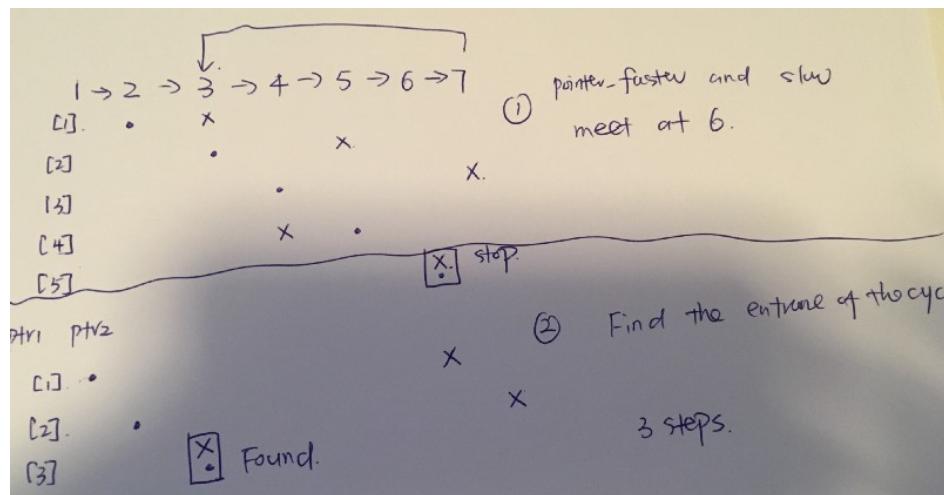


Figure 23.1: Example of floyd's cycle finding

```

1 def detectCycle(self, A):
2     #find the "intersection"
3     p_f=p_s=A
4     while (p_f and p_s and p_f.next):
5         p_f = p_f.next.next

```

```

6     p_s = p_s.next
7     if p_f==p_s:
8         break
9     #Find the "entrance" to the cycle.
10    ptr1 = A
11    ptr2 = p_s;
12    while ptr1 and ptr2:
13        if ptr1!=ptr2:
14            ptr1 = ptr1.next
15            ptr2 = ptr2.next
16        else:
17            return ptr1
18    return None

```

23.8 Exercise

23.8.1 Number

313. Super Ugly Number

¹ Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of size k. For example, [1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32] is the sequence of the first 12 super ugly numbers given primes = [2, 7, 13, 19] of size 4.

²

³ Note:

- ⁴ (1) 1 is a super ugly number for any given primes.
- ⁵ (2) The given numbers in primes are in ascending order.
- ⁶ (3) $0 < k \leq 100$, $0 < n \leq 106$, $0 < \text{primes}[i] < 1000$.
- ⁷ (4) The nth super ugly number is guaranteed to fit in a 32-bit signed integer.

```

1 def nthSuperUglyNumber(self, n, primes):
2     """
3         :type n: int
4         :type primes: List[int]
5         :rtype: int
6     """
7     nums=[1]
8     idexs=[0]*len(primes) #first is the current index
9     for i in range(n-1):
10         min_v = maxsize
11         min_j = []
12         for j, index in enumerate(idexs):
13             v = nums[index]*primes[j]
14             if v<min_v:
15                 min_v = v
16                 min_j=[j]
17             elif v==min_v:
18                 min_j.append(j) #we can get multiple j if
there is a tie

```

```
19     nums.append(min_v)
20     for j in min_j:
21         indexs[j] += 1
22     return nums[-1]
```

Part VII

Problem-Patterns

24

Array Questions(15%)

In this chapter, we mainly discuss about the array based questions. We first categorize these problems into different type, and then each type can usually be solved and optimized with nearly the best efficiency.

Given an array, a subsequence is composed of elements whose subscripts are increasing in the original array. A subarray is a subset of subsequence, which is contiguous subsequence. Subset contain any possible combinations of the original array. For example, for array [1, 2, 3, 4]:

```
Subsequence
[1, 3]
[1, 4]
[1, 2, 4]
Subarray
[1, 2]
[2, 3]
[2, 3, 4]
Subset includes different length of subset, either
length 0: []
length 1: [1], [2], [3], [4]
length 2: [1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]
```

Here array means one dimension list. For array problems, math will play an important role here. The rules are as follows:

- Subarray: using dynamic programming based algorithm to make brute force $O(n^3)$ to $O(n)$. Two pointers for the increasing subarray. Prefix sum, or kadane's algorithm plus sometimes with the hashmap, or two pointers (three pointers) for the maximum subarray.
- Subsequence: using dynamic programming based algorithm to make brute force $O(2^n)$ to $O(n^2)$, which corresponds to the seqence type of

dynamic programming.

- Duplicates: 217, 26, 27, 219, 287, 442;
- Intersections of Two Arrays:

Before we get into solving each type of problems, we first introduce the algorithms we will needed in this Chapter, including two pointers (three pointers or sliding window), prefix sum, kadane's algorithm. Kadane's algorithm can be explained with sequence type of dynamic programming.

After this chapter, we need to learn the step to solve these problems:

1. Analyze the problem and categorize it. To know the naive solution's time complexity can help us identify it.
2. If we can not find what type it is, let us see if we can *convert*. If not, we can try to identify a simple version of this problem, and then upgrade the simple solution to the more complex one.
3. Solve the problem with the algorithms we taught in this chapter.
4. Try to see if there is any more solutions.
5. Check the special case. (Usually very important for this type of problems)

24.1 Subarray

Note: For subarray the most important feature is contiguous. Here, we definitely will not use sorting. Given an array with size n , the total number of subarrays we have is $\sum_{i=1}^{i=n} i = n * (n + 1)/2$, which makes the time complexity of naive solution that use two nested for/while loop $O(n^2)$ or $O(n^3)$.

There are two types of problems related to subarry: **Range Query** and **optimization-based subarray**. The Range query problems include querying the minimum/maximum or sum of all elements in a given range $[i,j]$ in an array. Range Query has a more standard way to solve, either by searching or with the segment tree:

Range Query

1. 303. Range Sum Query - Immutable
2. 307. Range Sum Query - Mutable
3. 304. Range Sum Query 2D - Immutable

Optimization-based subarray Given a single array, we would normally be asked to return either the maximum/minimum value, the maximum/minimum length, or the number of subarrays that has sum/product that *satisfy a certain condition*. The condition here decide the difficulty of these problems.

The questions can be classified into two categories:

1. *Absolute-conditioned Subarray* that $\text{sum}/\text{product} = K$ or
2. *Vague-conditioned subarray* that has these symbols that is not equal.

With the proposed algorithms, the time complexity of subarray problems can be decreased from the brute force $O(n^3)$ to $O(n)$. The brute force is universal: two nested for loops marked the start and end of the subarray to enumerate all the possible subarrays, and another $O(n)$ spent to compute the result needed (sum or product or check the pattern like increasing or decreasing).

As we have discussed in the algorithm section,

1. **stack/queue/monotone stack** can be used to solve subarray problems that is related to its smaller/larger item to one item's left/right side
2. **sliding window** can be used to find subarray that either the sum or product inside of the sliding window is ordered (either monotone increasing/decreasing). This normally requires that the array are all positive or all negative. We can use the sliding window to cover its all search space. Or else we can't use sliding window.
3. For all problems related with subarray sum/product, for both vague or absolute conditioned algorithm, we have a universal algorithm: save the prefix sum (sometimes together with index) in a sorted array, and use binary search to find all possible starting point of the window.
4. Prefix Sum or Kadane's algorithm can be used when we need to get the sum of the subarray.
 1. 53. Maximum Subarray (medium)
 2. 325. Maximum Size Subarray Sum Equals k
 3. 525. Contiguous Array
 4. 560. Subarray Sum Equals K
 5. 209. Minimum Size Subarray Sum (medium)

Monotone stack and vague conditioned subarray

1. 713. Subarray Product Less Than K (all positive)
2. 862. Shortest Subarray with Sum at Least K (with negative)
3. 907. Sum of Subarray Minimums (all positive, but minimum in all subarray and sum)

24.1.1 Absolute-conditioned Subarray

For the maximum array, you are either asked to return:

1. the maximum sum or product; *solved using prefix sum or kadane's algorithm*
2. the maximum length of subarray with sum or product S equals to K; *solved using prefix sum together with a hashmap saves previous prefix sum and its indices*
3. the maximum number of subarray with sum or product S (the total number of) equals to K; *solved using prefix sum together with a hashmap saves previous prefix sum and its count*

Maximum/Minimum sum or product

24.1 53. Maximum Subarray (medium). Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$, the contiguous subarray $[4, -1, 2, 1]$ has the largest sum = 6.

Solution: Brute force is to use two for loops, first is the starting, second is the end, then we can get the maximum value. To optimize, we can use divide and conquer, $O(nlg n)$ vs brute force is $O(n^3)$ (two embedded for loops and n for computing the sum). The divide and conquer method was shown in that chapter. A more efficient algorithm is using pre_sum. Please check Section ?? for the answer.

Now what is the sliding window solution? The key step in sliding window is when to move the first pointer of the window (shrinking the window). The window must include current element j. For the maximum subarray, to increase the sum of the window, we need to abandon any previous elements if they have negative sum.

```

1 from sys import maxsize
2 class Solution:
3     def maxSubArray(self, nums):
4         """
5             :type nums: List[int]

```

```

6     :rtype: int
7     """
8     if not nums:
9         return 0
10    i, j = 0, 0 #i<=j
11    maxValue = -maxsize
12    window_sum = 0
13    while j < len(nums):
14        window_sum += nums[j]
15        j += 1
16        maxValue = max(maxValue, window_sum)
17        while i < j and window_sum < 0:
18            window_sum -= nums[i]
19            i += 1
20    return maxValue

```

Maximum/Minimum length of subarray with sum or product S

For this type of problem we need to track the length of it.

24.2 325. Maximum Size Subarray Sum Equals k.

Given an array `nums` and a target value `k`, find the maximum length of a subarray that sums to `k`. If there isn't one, return 0 instead. *Note: The sum of the entire `nums` array is guaranteed to fit within the 32-bit signed integer range.*

Example 1:

Given `nums = [1, -1, 5, -2, 3]`, `k = 3`,
 return 4. (because the subarray `[1, -1, 5, -2]` sums to 3
 and is the longest)

Example 2:

Given `nums = [-2, -1, 2, 1]`, `k = 1`,
 return 2. (because the subarray `[-1, 2]` sums to 1 and is
 the longest)

Follow Up:

Can you do it in $O(n)$ time?

Solution: Prefix Sum Saved as Hashmap. Answer: the brute force solution of this problem is the same as the maximum subarray. The similarity here is we track the prefix sum $S_{(i,j)} = y_j - y_{i-1}$, if we only need to track a certain value of $S_{(i,j)}$, which is k . Because $y_i = y_j - k$ which is the current prefix sum minus the k . If we use a hashmap to save the set of prefix sum together with the first index of this value appears. We saved $(y_i, \text{first_index})$, so that $\text{max_len} = \max(\text{idx} - \text{dict}[y_j - k])$.

```

1 def maxSubArrayLen(self, nums, k):
2     """

```

```

3     :type nums: List[int]
4     :type k: int
5     :rtype: int
6     """
7     prefix_sum = 0
8     dict = {0:-1} #this means for index -1, the sum is
9     0
10    max_len = 0
11    for idx, n in enumerate(nums):
12        prefix_sum += n
13        # save the set of prefix sum together with the
14        # first index of this value appears.
15        if prefix_sum not in dict:
16            dict[prefix_sum] = idx
17        # track the maximum length so far
18        if prefix_sum-k in dict:
            max_len=max(max_len, idx-dict[prefix_sum-k])
return max_len

```

Another example that asks for pattern but can be converted or equivalent to the last problems:

24.3 525. Contiguous Array. Given a binary array, find the maximum length of a contiguous subarray with equal number of 0 and 1. *Note: The length of the given binary array will not exceed 50,000.*

Example 1:

```

Input: [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with
equal number of 0 and 1.

```

Example 2:

```

Input: [0,1,0]
Output: 2
Explanation: [0, 1] (or [1, 0]) is a longest contiguous
subarray with equal number of 0 and 1.

```

Solution: the problem is similar to the maximum sum of array with sum==0, so 0=-1, 1==1. Here our $k = 0$

```

1 def findMaxLength(self, nums):
2     """
3     :type nums: List[int]
4     :rtype: int
5     """
6     nums=[nums[i] if nums[i]==1 else -1 for i in range(
7         len(nums))]
8     max_len=0
9     cur_sum=0
10    mapp={0:-1}

```

```

11
12     for idx,v in enumerate(nums):
13         cur_sum+=v
14         if cur_sum in mapp:
15             max_len=max(max_len, idx-mapp[cur_sum])
16         else:
17             mapp[cur_sum]=idx
18
19     return max_len

```

24.4 674. Longest Continuous Increasing Subsequence Given an unsorted array of integers, find the length of longest continuous increasing subsequence (subarray).

Example 1:

Input: [1,3,5,4,7]

Output: 3

Explanation: The longest continuous increasing subsequence is [1,3,5], its length is 3.

Even though [1,3,5,7] is also an increasing subsequence, it's not a continuous one where 5 and 7 are separated by 4.

Example 2:

Input: [2,2,2,2,2]

Output: 1

Explanation: The longest continuous increasing subsequence is [2], its length is 1.

\textit{Note: Length of the array will not exceed 10,000.}

Solution: The description of this problem should use "subarray" instead of the "subsequence". The brute force solution is like any subarray problem $O(n^3)$. For embedded for loops to enumerate the subarray, and another $O(n)$ to check if it is strictly increasing. Using two pointers, we can get $O(n)$ time complexity. We put two pointers: one i located at the first element of the nums, second j at the second element. We specifically restrict the subarray from i to j to be increasing, if this is violated, we reset the starting point of the subarray from the violated place.

```

1 class Solution:
2     def findLengthOfLCIS(self, nums):
3         """
4             :type nums: List[int]
5             :rtype: int
6         """
7         if not nums:
8             return 0
9         if len(nums)==1:
10            return 1
11        i,j = 0,0
12        max_length = 0

```

```

13     while j < len(nums):
14         j += 1 #slide the window
15         max_length = max(max_length, j-i)
16         # when condition violated, reset the window
17         if j<len(nums) and nums[j-1]>=nums[j]:
18             i = j
19
20     return max_length

```

24.5 209. Minimum Size Subarray Sum (medium) Given an array of n positive integers and a positive integer s , find the minimal length of a contiguous subarray of which the sum $\geq s$. If there isn't one, return 0 instead.

Example :

```

Input: s = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: the subarray [4,3] has the minimal length
under the problem constraint.

```

Solution 1: Sliding Window, $O(n)$.

```

1 def minSubArrayLen(self, s, nums):
2     ans = float('inf')
3     n = len(nums)
4     i = j = 0
5     acc = 0
6     while j < n:
7         acc += nums[j] # increase the window size
8         while acc >= s: # shrink the window to get the
9             optimal result
10            ans = min(ans, j-i+1)
11            acc -= nums[i]
12            i += 1
13        j += 1
14    return ans if ans != float('inf') else 0

```

Solution 2: prefix sum and binary search. $O(n \log n)$. Assuming current prefix sum is p_i , We need to find the max $p_j \leq (p_i - s)$, this is the right most value in the prefix sum array (sorted) that is $\leq p_i - s$.

```

1 from bisect import bisect_right
2 class Solution(object):
3     def minSubArrayLen(self, s, nums):
4         ans = float('inf')
5         n = len(nums)
6         i = j = 0
7         ps = [0]
8         while j < n:
9             ps.append(nums[j]+ps[-1])
10            # find a possible left i
11            if ps[-1]-s >= 0:

```

```

12         index = bisect_right(ps, ps[-1]-s)
13         if index > 0:
14             index -= 1
15             ans = min(ans, j-index+1)
16             j+=1
17     return ans if ans != float('inf') else 0

```

The maximum number of subarray with sum or product S

24.6 560. Subarray Sum Equals K Given an array of integers and an integer k, you need to find the total number of continuous subarrays whose sum equals to k.

Example 1:
Input: nums = [1, 1, 1], k = 2
Output: 2

Answer: The naive solution is we enumerate all possible subarray which is n^2 , and then we compute and check its sum which is $O(n)$. So the total time complexity is $O(n^3)$ time complexity. However, we can decrease it to $O(n^2)$ if we compute the sum till current index for each position, with equation $sum(i, j) = sum(0, j) - sum(0, i)$. However the OJ gave us LTE error.

```

1 def subarraySum(self, nums, k):
2     """
3         :type nums: List[int]
4         :type k: int
5         :rtype: int
6     """
7     """ return the number of subarrays that equal to k
8     """
9     count = 0
10    sums = [0]*(len(nums)+1) # sum till current index
11    for idx, v in enumerate(nums):
12        sums[idx+1] = sums[idx]+v
13    for i in range(len(nums)):
14        for j in range(i, len(nums)):
15            value = sums[j+1]-sums[i]
16            count = count+1 if value==k else count
17    return count

```

Solution 3: using prefix_sum and hashmap, to just need to reformulate dict[sum_i]. For this question, we need to get the total number of subsubarray, so $dict[i] = count$, which means every time we just set the dict[i]+=1. dict[0]=1

```

1 import collections
2 class Solution(object):
3     def subarraySum(self, nums, k):

```

```

4      """
5      :type nums: List[int]
6      :type k: int
7      :rtype: int
8      """
9      ''' return the number of subarrays that equal to k
10     '''
11     dict = collections.defaultdict(int) #the value is
12     the number of the sum occurs
13     dict[0]=1
14     prefix_sum, count=0, 0
15     for v in nums:
16         prefix_sum += v
17         count += dict[prefix_sum-k] # increase the
18         counter of the appearing value k, default is 0
19         dict[prefix_sum] += 1 # update the count of
20         prefix sum, if it is first time, the default value is 0
21     return count

```

- 24.7 974. Subarray Sums Divisible by K.** Given an array A of integers, return the number of (contiguous, non-empty) subarrays that have a sum divisible by K.

Example 1:

Input: A = [4,5,0,-2,-3,1], K = 5

Output: 7

Explanation: There are 7 subarrays with a sum divisible by K = 5:

[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

Analysis: for the above array, we can compute the prefix sum as [0,4,9, 9, 7,4,5]. Let $P[i+1] = A[0] + A[1] + \dots + A[i]$. Then, each subarray can be written as $P[j] - P[i]$ (for $j > i$). We need to find for current j index that $(P[j]-P[i]) \% K == 0$. Because $P[j]\%K=P[i]\%K$, therefore different compared with when sum == K, we not check $P[j]\%K$ but instead $P[j]\%K$ if it is in the hashmap. Therefore, we need to save the prefix sum as the modulo of K. For the example, we have dict: 0: 2, 4: 4, 2: 1.

```

1 from collections import defaultdict
2 class Solution:
3     def subarraysDivByK(self, A, K):
4         """
5         :type A: List[int]
6         :type K: int
7         :rtype: int
8         """
9         a_sum = 0
10        p_dict = defaultdict(int)
11        p_dict[0] = 1 # when it is empty we still has one
12        0:1

```

```

12     ans = 0
13     for i, v in enumerate(A):
14         a_sum += v
15         a_sum %= K
16         if a_sum in p_dict:
17             ans += p_dict[a_sum]
18             p_dict[a_sum] += 1 # save the remodule instead
19     return ans

```

Solution 2: use Combination Then $P = [0,4,9,9,7,4,5]$, and $C_0 = 2, C_2 = 1, C_4 = 4$. With $C_0 = 2$, (at $P[0]$ and $P[6]$), it indicates C_2^1 subarray with sum divisible by K , namely $A[0:6]=[4, 5, 0, -2,-3,1]$. With $C_4 = 4$ (at $P[1], P[2], P[3], P[5]$), it indicates $C_4^2 = 6$ subarrays with sum divisible by K , namely $A[1:2], A[1:3], A[1:5], A[2:3], A[2:5], A[3:5]$.

```

1 def subarraysDivByK(self, A, K):
2     P = [0]
3     for x in A:
4         P.append((P[-1] + x) % K)
5
6     count = collections.Counter(P)
7     return sum(v*(v-1)/2 for v in count.values())

```

24.1.2 Vague-conditioned subarray

In this section, we would be asked to ask the same type of question compared with the last section. The only difference is the condition. For example, in the following question, it is asked with subarray that with $sum \geq s$.

Because of the vague of the condition, a hashmap+prefix sum solution will no longer give us $O(n)$ linear time. The best we can do if the array is all positive number we can gain $O(nlgn)$ if it is combined with binary search. However, a carefully designed sliding window can still help us achieve linear time $O(n)$. For array with negative number, we can utilize monotonic queue mentioned in Section 21.1, which will achieve $O(n)$ both in time and space complexity.

All Positive Array (Sliding Window) If it is all positive array, it can still be easily solved with sliding window. For example:

24.8 209. Minimum Size Subarray Sum (medium) Given an array of n positive integers and a positive integer s , find the minimal length of a contiguous subarray of which the sum $\geq s$. If there isn't one, return 0 instead.

Example :

Input: $s = 7$, $nums = [2, 3, 1, 2, 4, 3]$

Output: 2

Explanation: the subarray $[4, 3]$ has the minimal length under the problem constraint .

Follow up: If you have figured out the $O(n)$ solution, try coding another solution of which the time complexity is $O(n \log n)$.

Analysis. For this problem, we can still use prefix sum saved in hashmap. However, since the condition is $sum \geq s$, if we use a hashmap, we need to search through the hashmap with $key \leq prefixSum - s$. The time complexity would rise up to $O(n^2)$ if we use linear search. We would receive LTE error.

```

1  def minSubArrayLen(self, s, nums):
2      """
3          :type s: int
4          :type nums: List[int]
5          :rtype: int
6      """
7      if not nums:
8          return 0
9      dict = collections.defaultdict(int)
10     dict[0] = -1 # pre_sum 0 with index -1
11     prefixSum = 0
12     minLen = sys.maxsize
13     for idx, n in enumerate(nums):
14         prefixSum += n
15         for key, value in dict.items():
16             if key <= prefixSum - s:
17                 minLen = min(minLen, idx - value)
18         dict[prefixSum] = idx #save the last index
19     return minLen if 1<=minLen<=len(nums) else 0

```

Solution 1: Prefix Sum and Binary Search. Because the items in the array are all positive number, so the prefix sum array is increasing, this means if we save the prefix sum in an array, it is ordered, we can use binary search to find the index of largest value $\leq (prefixSum - s)$. If we use bisect module, we can use bisect_right function which finds the right most position that we insert current value to keep the array ordered. The index will be $r-1$.

```

1  import bisect
2  def minSubArrayLen(self, s, nums):
3      ps = [0]
4      ans = len(nums)+1
5      for i, v in enumerate(nums):
6          ps.append(ps[-1] + v)
7          #find the right most position that <=
8          rr = bisect.bisect_right(ps, ps[i+1] - s)
9          if rr:
10             ans = min(ans, i+1 - (rr-1))
11     return ans if ans <= len(nums) else 0

```

```

1  def minSubArrayLen(self, s, nums):
2      """
3          :type s: int
4          :type nums: List[int]

```

```

5      :rtype: int
6      """
7      def bSearch(nums, i, j, target):
8          while i < j:
9              mid = (i+j) / 2
10             if nums[mid] == target:
11                 return mid
12             elif nums[mid] < target:
13                 i = mid + 1
14             else:
15                 j = mid - 1
16             return i
17
18         if not nums:
19             return 0
20         rec = [0] * len(nums)
21         rec[0] = nums[0]
22         if rec[0] >= s:
23             return 1
24         minlen = len(nums)+1
25         for i in range(1, len(nums)):
26             rec[i] = rec[i-1] + nums[i]
27             if rec[i] >= s:
28                 index = bSearch(rec, 0, i, rec[i] - s)
29                 if rec[index] > rec[i] - s:
30                     index -= 1
31                 minlen = min(minlen, i - index)
32         return minlen if minlen != len(nums)+1 else 0

```

Solution 2: Sliding window in $O(n)$. While, using the sliding window, Once the sum in the window satisfy the condition, we keep shrinking the window size (moving the left pointer rightward) until the condition is no longer hold. This way, we are capable of getting the complexity with $O(n)$.

```

1 def minSubArrayLen(self, s, nums):
2     i, j = 0, 0
3     sum_in_window = 0
4     ans = len(nums) + 1
5     while j < len(nums):
6         sum_in_window += nums[j]
7         j += 1
8         # shrink the window if the condition satisfied
9         while i < j and sum_in_window >= s:
10             ans = min(ans, j-i)
11             sum_in_window -= nums[i]
12             i += 1
13     return ans if ans <= len(nums) else 0

```

24.9 713. Subarray Product Less Than K Your are given an array of positive integers nums. Count and print the number of (contiguous) subarrays where the product of all the elements in the subarray is less

than k.

Example 1:

Input: nums = [10, 5, 2, 6], k = 100

Output: 8

Explanation: The 8 subarrays that have product less than 100 are: [10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6].

Note that [10, 5, 2] is not included as the product of 100 is not strictly less than k.

Note:

$0 < \text{nums.length} \leq 50000$.

$0 < \text{nums}[i] < 1000$.

$0 \leq k < 10^6$.

Answer: Because we need the subarray less than k, so it is difficult to use prefix sum. If we use sliding window,

```

1 i=0, j=0, 10 10<100, ans+= j-i+1 (1) -> [10]
2 i=0, j=1, 50 50<100, ans+= j-i+1 (3), -> [10],[10,5]
3 i=0, j=2, 100 shrink the window, i=1, product = 10, ans+=2,
   ->[5,2][2]
4 i=1, j=3, 60, ans+=3 ->[2,6],[2],[6]
```

The python code:

```

1 class Solution:
2     def numSubarrayProductLessThanK(self, nums, k):
3         """
4             :type nums: List[int]
5             :type k: int
6             :rtype: int
7         """
8         if not nums:
9             return 0
10        i, j = 0, 0
11        window_product = 1
12        ans = 0
13        while j < len(nums):
14            window_product *= nums[j]
15
16            while i < j and window_product >= k:
17                window_product /= nums[i]
18                i += 1
19            if window_product < k:
20                ans += j - i + 1
21                j += 1
22        return ans
```

Array with Negative Element (Monotonic Queue) In this section, we will work through how to handle the array with negative element and is Vague-conditioned. We found using monotonic Queue or stack (Section 21.1

will fit the scenario and gave $O(n)$ time complexity and $O(N)$ space complexity.

24.10 862. Shortest Subarray with Sum at Least K Return the length of the shortest, non-empty, contiguous subarray of A with sum at least K.

If there is no non-empty subarray with sum at least K, return -1.

Example 1:

Input: A = [1], K = 1
Output: 1

Example 2:

Input: A = [1, 2], K = 4
Output: -1

Example 3:

Input: A = [2, -1, 2], K = 3
Output: 3

Note: $1 \leq A.length \leq 50000$, $-10^5 \leq A[i] \leq 10^5$, $1 \leq K \leq 10^9$.

Analysis: The only difference of this problem compared with the last is with negative value. Because of the negative, the shrinking method no longer works because when we shrink the window, the sum in the smaller window might even grow if we just cut out a negative value. For instance, [84, -37, 32, 40, 95], K=167, the right answer is [32, 40, 95]. In this program, i=0, j=4, so how to handle the negative value?

Solution 1: prefix sum and binary search in prefix sum. LTE

```

1 def shortestSubarray(self, A, K):
2     def bisect_right(lst, target):
3         l, r = 0, len(lst)-1
4         while l <= r:
5             mid = l + (r-l)//2
6             if lst[mid][0] <= target:
7                 l = mid + 1
8             else:
9                 r = mid - 1
10        return l
11    acc = 0
12    ans = float('inf')
13    prefixSum = [(0, -1)] #value and index
14    for i, n in enumerate(A):
15        acc += n
16        index = bisect_right(prefixSum, acc-K)
17        for j in range(index):
18            ans = min(ans, i-prefixSum[j][1])
19        index = bisect_right(prefixSum, acc)
20        prefixSum.insert(index, (acc, i))

```

```

21     #print(index, prefixSum)
22     return ans if ans != float('inf') else -1

```

Now, let us analyze a simple example which includes both 0 and negative number. [2, -1, 2, 0, 1], K=3, with prefix sum [0, 2, 1, 3, 3, 4], the subarray is [2,-1,2], [2,-1,2, 0] and [2, 0, 1] where its sum is at least three. First, let us draw the prefix sum on a x-y axis. When we encounter an negative number, the prefix sum decreases, if it is zero, then the prefix sum stabilize. For the zero case: at p[2] = p[3], if subarray ends with index 2 is considered, then 3 is not needed. For the negative case: p[0]=2>p[1]=1 due to A[1]<0. Because p[1] can always be a better choice to be i than p[1] (smaller so that it is more likely, shorter distance). Therefore, we can still keep the validate prefix sum monotonically increasing like the array with all positive numbers by maintaining a mono queue.

```

1 class Solution:
2     def shortestSubarray(self, A, K):
3
4         P = [0]*(len(A)+1)
5         for idx, x in enumerate(A):
6             P[idx+1] = P[idx]+x
7
8
9         ans = len(A)+1 # N+1 is impossible
10        monoq = collections.deque()
11        for y, Py in enumerate(P):
12            while monoq and Py <= P[monoq[-1]]: #both
negative and zero leads to kick out any previous larger
or equal value
13                print('pop', P[monoq[-1]])
14                monoq.pop()
15
16            while monoq and Py - P[monoq[0]] >= K: # if one
x is considered, no need to consider again (similar to
sliding window where we move the first index forward)
17                print('pop', P[monoq[0]])
18                ans = min(ans, y - monoq.popleft())
19            print('append', P[y])
20            monoq.append(y)
21
22
23        return ans if ans < len(A)+1 else -1

```

24.1.3 LeetCode Problems and Misc

Absolute-conditioned Subarray

1. 930. Binary Subarrays With Sum

```

1     In an array A of 0s and 1s, how many non-empty
subarrays have sum S?

```

```

2 Example 1:
3
4 Input: A = [1,0,1,0,1], S = 2
5 Output: 4
6 Explanation:
7 The 4 subarrays are bolded below:
8 [1,0,1,0,1]
9 [1,0,1,0,1]
10 [1,0,1,0,1]
11 [1,0,1,0,1]
12 Note:
13
14     A.length <= 30000
15     0 <= S <= A.length
16     A[i] is either 0 or 1.

```

Answer: this is exactly the third time of maximum subarray, the maximum length of subarry with a certain value. We solve it using prefix sum and a hashmap to save the count of each value.

```

1 import collections
2 class Solution:
3     def numSubarraysWithSum(self, A, S):
4         """
5             :type A: List[int]
6             :type S: int
7             :rtype: int
8         """
9         dict = collections.defaultdict(int) #the value is
10        the number of the sum occurs
11        dict[0]=1 #prefix sum starts from 0 and the number
12        is 1
13        prefix_sum, count=0, 0
14        for v in A:
15            prefix_sum += v
16            count += dict[prefix_sum-S] # increase the
counter of the appearing value k, default is 0
17            dict[prefix_sum] += 1 # update the count of
prefix sum, if it is first time, the default value is 0
18        return count

```

We can write it as:

```

1     def numSubarraysWithSum(self, A, S):
2         """
3             :type A: List[int]
4             :type S: int
5             :rtype: int
6         """
7         P = [0]
8         for x in A: P.append(P[-1] + x)
9         count = collections.Counter()
10
11         ans = 0

```

```

12     for x in P:
13         ans += count[x]
14         count[x + S] += 1
15
16     return ans

```

Also, it can be solved used a modified sliding window algorithm. For sliding window, we have i, j starts from 0, which represents the window. Each iteration j will move one position. For a normal sliding window, only if the sum is larger than the value, then we shrink the window size by one. However, in this case, like in the example 1, 0, 1, 0, 1, when $j = 5$, $i = 1$, the sum is 2, but the algorithm would miss the case of $i = 2$, which has the same sum value. To solve this problem, we keep another index i_{hi} , in addition to the moving rule of i , it also moves if the sum is satisfied and that value is 0. This is actually a Three pointer algorithm.

```

1 def numSubarraysWithSum(self, A, S):
2     i_lo, i_hi, j = 0, 0, 0 #i_lo <= j
3     sum_lo = sum_hi = 0
4     ans = 0
5     while j < len(A):
6         # Maintain i_lo, sum_lo:
7         # While the sum is too big, i_lo += 1
8         sum_lo += A[j]
9         while i_lo < j and sum_lo > S:
10            sum_lo -= A[i_lo]
11            i_lo += 1
12
13         # Maintain i_hi, sum_hi:
14         # While the sum is too big, or equal and we can
15         # move, i_hi += 1
16         sum_hi += A[j]
17         while i_hi < j and (
18             sum_hi > S or sum_hi == S and not A[
19             i_hi]):
20             sum_hi -= A[i_hi]
21             i_hi += 1
22
23         if sum_lo == S:
24             ans += i_hi - i_lo + 1
25
26     return ans

```

2. 523. Continuous Subarray Sum

¹ Given a list of non-negative numbers and a target integer k , write a function to check if the array has a continuous subarray of size at least 2 that sums up to the multiple of k , that is, sums up to $n*k$ where n is also an integer.

```

2
3 Example 1:
4 Input: [23, 2, 4, 6, 7], k=6
5 Output: True
6 Explanation: Because [2, 4] is a continuous subarray of
    size 2 and sums up to 6.
7
8 Example 2:
9 Input: [23, 2, 6, 4, 7], k=6
10 Output: True
11 Explanation: Because [23, 2, 6, 4, 7] is an continuous
    subarray of size 5 and sums up to 42.
12
13 Note:
14 The length of the array won't exceed 10,000.
15 You may assume the sum of all the numbers is in the range
    of a signed 32-bit integer.

```

Answer: This is a mutant of the subarray with value k. The difference here, we save the prefix sum as the remainder of k. if $(a + b) \% k = 0$, then $(a \% k + b \% k) / k = 1$.

```

1 class Solution:
2     def checkSubarraySum(self, nums, k):
3         """
4             :type nums: List[int]
5             :type k: int
6             :rtype: bool
7         """
8
9         if not nums:
10             return False
11         k = abs(k)
12         prefixSum = 0
13         dict = collections.defaultdict(int)
14         dict[0]=-1
15         for i, v in enumerate(nums):
16             prefixSum += v
17             if k!=0:
18                 prefixSum %= k
19             if prefixSum in dict and (i-dict[prefixSum])
>=2:
20                 return True
21             if prefixSum not in dict:
22                 dict[prefixSum] = i
23         return False

```

For problems like bounded, or average, minimum in a subarray,

24.11 795.Number of Subarrays with Bounded Maximum (medium)

24.12 907. Sum of Subarray Minimums (monotone stack)

24.2 Subsequence (Medium or Hard)

The difference of the subsequence type of questions with the subarray is that we do not need the elements to be consecutive. Because of this relaxation, the brute force solution of this type of question is exponential $O(2^n)$, because for each element, we have two options: chosen or not chosen. This type of questions would usually be used as a follow-up question to the subarray due to its further difficulty because of nonconsecutive. This type of problems are a typical dynamic programming. Here we should a list of all related subsequence problems shown on LeetCode in Fig. 24.1

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not). For the subsequence problems, commonly we will see increasing subsequence, count the distinct subsequence. And they are usually solved with single sequence type of dynamic programming. 940. Distinct Subsequences II (hard)

#	Title	Solution	Acceptance	Difficulty	Frequency
3	Longest Substring Without Repeating Characters	25.1%	Medium	Medium	High
300	Longest Increasing Subsequence	39.4%	Medium	Medium	Medium
674	Longest Continuous Increasing Subsequence	43.0%	Easy	Easy	Medium
891	Sum of Subsequence Widths	25.4%	Hard	Hard	Low
521	Longest Uncommon Subsequence I	55.7%	Easy	Easy	Medium
659	Split Array into Consecutive Subsequences	37.7%	Medium	Medium	Medium
516	Longest Palindromic Subsequence	43.9%	Medium	Medium	Medium
792	Number of Matching Subsequences	38.8%	Medium	Medium	Medium
673	Number of Longest Increasing Subsequence	32.2%	Medium	Medium	Medium
727	Minimum Window Subsequence	33.1%	Hard	Hard	Medium
730	Count Different Palindromic Subsequences	37.4%	Hard	Hard	Medium
392	Is Subsequence	45.2%	Medium	Medium	Medium
491	Increasing Subsequences	40.1%	Medium	Medium	Medium
873	Length of Longest Fibonacci Subsequence	42.1%	Medium	Medium	Medium
334	Increasing Triplet Subsequence	39.3%	Medium	Medium	Medium
446	Arithmetic Slices II - Subsequence	28.5%	Hard	Hard	Medium
456	132 Pattern	27.3%	Medium	Medium	Medium
376	Wiggle Subsequence	36.3%	Medium	Medium	Medium
444	Sequence Reconstruction	19.7%	Medium	Medium	Medium
115	Distinct Subsequences	33.5%	Hard	Hard	Medium
594	Longest Harmonious Subsequence	41.8%	Easy	Easy	Medium
522	Longest Uncommon Subsequence II	32.3%	Medium	Medium	Medium

Figure 24.1: Subsequence Problems Listed on LeetCode

Given a string S , count the number of distinct, non-empty subsequences of S . Since the result may be large, return the answer modulo $10^9 + 7$.

```

1 Example 1:
2
3 Input: "abc"
4 Output: 7
5 Explanation: The 7 distinct subsequences are "a", "b", "c", "ab"
6   ", "ac", "bc", and "abc".
7 Example 2:
8
9 Input: "aba"
10 Output: 6
11 Explanation: The 6 distinct subsequences are "a", "b", "ab", "ba"
12   ", "aa" and "aba".
13 Example 3:
14
15 Input: "aaa"
16 Output: 3
17 Explanation: The 3 distinct subsequences are "a", "aa" and "aaa"
18   ".

```

Sequence type dynamic programming. The naive solution for subsequence is using DFS to generate all of the subsequence recursively and we also need to check the repetition. The possible number of subsequence is $2^n - 1$. Let's try forward induction method.

```

1 # define the result for each state: number of subsequence ends
2   with each state
3 state: a   b   c
4 ans  : 1   2   4
5 a: a; dp[0] = 1
6 b: b, ab; = dp[0]+1 if this is 'a', length 1 is the same as dp
7   [0], only length 2 is possible
8 c: c, ac, bc, abc; = dp[0]+dp[1]+1, if it is 'a', aa, ba, aba,
9   = dp[1]+1
10 d: d, ad, bd, abd, cd, acd, bcd, abcd = dp[0]+dp[1]+dp[2]+1

```

Thus the recurrence function can be Eq. 24.1.

$$dp[i] = \sum_{j < i} (dp[j]) + 1, S[j]! = S[i] \quad (24.1)$$

Thus, we have $O(n^2)$ time complexity, and the following code:

```

1 def distinctSubseqII(self, S):
2     """
3     :type S: str
4     :rtype: int
5     """
6     MOD = 10**9+7
7     dp = [1]*len(S) #means for that length it has at least one
8     count
9     for i, c in enumerate(S):
10         for j in range(i):

```

```

10         if c == S[j]:
11             continue
12         else:
13             dp[i] += dp[j]
14             dp[i] %= MOD
15     return sum(dp) % MOD

```

However, we still get LTE. How to improve it further. If we use a counter indexed by all of the 26 letters, and a prefix sum. The inner for loop can be replaced by $dp[i] = 1 + (\text{prefix sum} - \text{sum of all } S[i])$. Thus we can lower the complexity further to $O(n)$.

```

1 def distinctSubseqII(self, S):
2     MOD = 10**9+7
3     dp = [1]*len(S) #means for that length it has at least one
4     count
5     sum_tracker = [0]*26
6     total = 0
7     for i, c in enumerate(S):
8         index = ord(c) - ord('a')
9         dp[i] += total-sum_tracker[index]
10        total += dp[i]
11        sum_tracker[index] += dp[i]
12    return sum(dp) % MOD

```

24.2.1 Others

For example, the following question would be used as follow up for question *Longest Continuous Increasing Subsequence*

300. Longest Increasing Subsequence

673. Number of Longest Increasing Subsequence

Given an unsorted array of integers, find the number of longest increasing subsequence.

```

1 Example 1:
2
3 Input: [1,3,5,4,7]
4 Output: 2
5 Explanation: The two longest increasing subsequence are [1, 3,
6           4, 7] and [1, 3, 5, 7].
7
8 Example 2:
9 Input: [2,2,2,2,2]
10 Output: 5
11 Explanation: The length of longest continuous increasing
12   subsequence is 1, and there are 5 subsequences' length is 1,
13   so output 5.
14 \textit{Note: Length of the given array will be not exceed 2000
15   and the answer is guaranteed to be fit in 32-bit signed int.}

```

Solution: Another different problem, to count the number of the max subsequence. Typical dp:

state: $f[i]$

```

1 from sys import maxsize
2 class Solution:
3     def findNumberOfLIS(self, nums):
4         """
5             :type nums: List[int]
6             :rtype: int
7         """
8         max_count = 0
9         if not nums:
10             return 0
11         memo =[None for _ in range(len(nums))]
12         rlst = []
13         def recursive(idx, tail, res):
14             if idx==len(nums):
15                 rlst.append(res)
16                 return 0
17             if memo[idx]==None:
18                 length = 0
19                 if nums[idx]>tail:
20                     addLen = 1+recursive(idx+1, nums[idx], res+[nums[idx]])
21                     notAddLen = recursive(idx+1, tail, res)
22                     return max(addLen, notAddLen)
23                 else:
24                     return recursive(idx+1, tail, res)
25
26
27         ans=recursive(0,-maxsize,[])
28         count=0
29         for lst in rlst:
30             if len(lst)==ans:
31                 count+=1
32
33         return count

```

Using dynamic programming, the difference is we add a count array.

```

1 from sys import maxsize
2 class Solution:
3     def findNumberOfLIS(self, nums):
4         N = len(nums)
5         if N <= 1: return N
6         lengths = [0] * N #lengths[i] = longest ending in nums[i]
7         counts = [1] * N #count[i] = number of longest ending in
8         #nums[i]
9
10        for idx, num in enumerate(nums): #i
11            for i in range(idx): #j
12                if nums[i] < nums[idx]: #bigger
13                    if lengths[i] >= lengths[idx]:
14                        lengths[idx] = 1 + lengths[i] #set the
15                        biggest length

```

```

14             counts[idx] = counts[i] #change the
15     count
16         elif lengths[i] + 1 == lengths[idx]: #if it
17             is a tie
18                 counts[idx] += counts[i] #increase the
19             current count by count[i]
20
21 longest = max(lengths)
22     print(counts)
23     print(lengths)
24     return sum(c for i, c in enumerate(counts) if lengths[i]
25             == longest)

```

128. Longest Consecutive Sequence

```

1 Given an unsorted array of integers , find the length of the
2 longest consecutive elements sequence.
3
4 For example ,
5 Given [100, 4, 200, 1, 3, 2],
6 The longest consecutive elements sequence is [1, 2, 3, 4].
7 Return its length: 4.
8
9 Your algorithm should run in O(n) complexity .

```

Solution: Not thinking about the $O(n)$ complexity, we can use sorting to get [1,2,3,4,100,200], and then use two pointers to get [1,2,3,4].

How about $O(n)$? We can pop out a number in the list, example, 4 , then we use while first-1 to get any number that is on the left side of 4, here it is 3, 2, 1, and use another to find all the bigger one and remove these numbers from the nums array.

```

1 def longestConsecutive(self , nums):
2     nums = set(nums)
3     maxlen = 0
4     while nums:
5         first = last = nums.pop()
6         while first - 1 in nums: #keep finding the smaller
7             one
8                 first -= 1
9                 nums.remove(first)
10            while last + 1 in nums: #keep finding the larger one
11                last += 1
12                nums.remove(last)
13            maxlen = max(maxlen, last - first + 1)
14
15        return maxlen

```

24.3 Subset(Combination and Permutation)

The Subset B of a set A is defined as a set within all elements of this subset are from set A. In other words, the subset B is contained inside the set A,

$B \in A$. There are two kinds of subsets: if the order of the subset doesn't matter, it is a combination problem, otherwise, it is a permutation problem. To solve the problems in this section, we need to refer to the backtracking in Sec ???. When the subset has a fixed constant length, then hashmap can be used to lower the complexity by one power of n.

Subset VS Subsequence. In the subsequence, the elements keep the original order from the original sequence. While, in the set concept, there is no ordering, only a set of elements.

In this type of questions, we are asked to return subsets of a list. For this type of questions, backtracking ?? can be applied.

24.3.1 Combination

The solution of this section is heavily correlated to Section ???. 78. Subsets

```

1 Given a set of distinct integers , nums, return all possible
2   subsets (the power set).
3 Note: The solution set must not contain duplicate subsets.
4
5 Example:
6
7 Input: nums = [1 ,2 ,3]
8 Output:
9 [
10   [3] ,
11   [1] ,
12   [2] ,
13   [1 ,2 ,3] ,
14   [1 ,3] ,
15   [2 ,3] ,
16   [1 ,2] ,
17   []
18 ]
```

Backtracking. This is a combination problem, which we have explained in backtrack section. We just directly gave the code here.

```

1 def subsets(self ,  nums):
2     res ,  n = [] ,  len(nums)
3     res = self.combine(nums,  n,  n)
4     return res
5
6 def combine( self ,  nums,  n,  k):
7     """
8         :type n: int
9         :type k: int
10        :rtype: List[List[int]]
11    """
12    def C_n_k(d,  k,  s,  curr ,  ans): #d controls the degree (depth
13        , k is controls the return level , curr saves the current
14        result ,  ans is all the result
```

```

13     ans.append(curr)
14     if d == k: #the length is satisfied
15
16         return
17     for i in range(s, n):
18         curr.append(nums[i])
19         C_n_k(d+1, k, i+1, curr[:], ans) # i+1 because no
repeat, make sure use deep copy curr[:]
20         curr.pop()
21
22     ans = []
23     C_n_k(0, k, 0, [], ans)
24     return ans

```

Incremental. Backtracking is not the only way for the above problem. There is another way to do it iterative, observe the following process. We can just keep append elements to the end of previous results.

```

1 [1, 2, 3, 4]
2 l = 0, []
3 l = 1, for 1, []+[1], -> [1], get powerset of [1]
4 l = 2, for 2, []+[2], [1]+[2], -> [2], [1, 2], get powerset of
[1, 2]
5 l = 3, for 3, []+[3], [1]+[3], [2]+[3], [1, 2]+[3], -> [3], [1,
3], [2, 3], [1, 2, 3], get powerset of [1, 2, 3]
6 l = 4, for 4, []+[4]; [1]+[4]; [2]+[4], [1, 2] +[4]; [3]+[4],
[1,3]+[4],[2,3]+[4], [1,2,3]+[4], get powerset of [1, 2, 3,
4]

```

```

1 def subsets(self, nums):
2     result = [[]] #use two dimensional, which already have []
one element
3     for num in nums:
4         new_results = []
5         for r in result:
6             new_results.append(r + [num])
7         result += new_results
8
9     return result

```

90. Subsets II

```

1 Given a collection of integers that might contain duplicates ,
nums, return all possible subsets (the power set).
2
3 Note: The solution set must not contain duplicate subsets.
4
5 Example:
6
7 Input: [1,2,2]
8 Output:
9 [
10   [2],
11   [1],
12   [1,2,2],

```

```

13     [2 ,2] ,
14     [1 ,2] ,
15     []
16 ]

```

Analysis: Because of the duplicates, the previous superset algorithm would give repetitive subset. For the above example, we would have [1, 2] twice, and [2] twice. If we try to modify on the previous code. We first need to sort the nums, which makes the way we check repeat easier. Then the code goes like this:

```

1 def subsetsWithDup( self ,  nums):
2     """
3         :type  nums:  List [ int ]
4         :rtype:  List [ List [ int ] ]
5     """
6
7     nums . sort ()
8     result = [ [] ] #use two dimensional , which already have
9     [ ] one element
10    for num in nums:
11        new_results = []
12        for r in result:
13            print(r)
14            new_results.append(r + [num])
15        for rst in new_results:
16            if rst not in result: # check the repetitive
17                result.append(rst)
18
19    return result

```

However, the above code is extremely inefficient because of the checking process. A better way to do this:

```

1 [1 , 2, 2]
2 l = 0, []
3 l = 1, for 1, []+[1]
4 l = 2, for 2, []+[2] , [1]+[2]; []+[2, 2], [1]+[2, 2]

```

So it would be more efficient if we first save all the numbers in the array in a dictionary. For the above case, the dic = 1:1, 2:2. Each time we try to generate the result, we use 2 up to 2 times. Same way, we can use dictionary on the backtracking too.

```

1 class Solution ( object ):
2     def subsetsWithDup( self ,  nums):
3         """
4             :type  nums:  List [ int ]
5             :rtype:  List [ List [ int ] ]
6         """
7
8         if not nums:
9             return [ [] ]
10        res = [ [] ]
11        dic = collections.Counter(nums)
12        for key , val in dic.items():

```

```

12     tmp = []
13     for lst in res:
14         for i in range(1, val+1):
15             tmp.append(lst+[key]*i)
16     res += tmp
17     return res

```

77. Combinations

```

1 Given two integers n and k, return all possible combinations of
2   k numbers out of 1 ... n.
3
4 Example:
5 Input: n = 4, k = 2
6 Output:
7 [
8   [2,4],
9   [3,4],
10  [2,3],
11  [1,2],
12  [1,3],
13  [1,4],
14 ]

```

Analysis: In this problem, it is difficult for us to generate the results iteratively, the only way we can use the second solution is by filtering and get only the results with the length we want. However, the backtrack can solve the problem easily as we mentioned in Section ??.

```

1 def combine(self, n, k):
2     """
3     :type n: int
4     :type k: int
5     :rtype: List[List[int]]
6     """
7     ans = []
8     def C_n_k(d, k, s, curr):
9         if d==k:
10             ans.append(curr)
11             return
12         for i in range(s, n):
13             #curr.append(i+1)
14             #C_n_k(d+1, k, i+1, curr[:])
15             #curr.pop()
16             C_n_k(d+1, k, i+1, curr+[i+1])
17             C_n_k(0, k, 0, [])
18
19     return ans

```

24.3.2 Combination Sum

39. Combination Sum

Given a set of candidate numbers (candidates) (**without duplicates**) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.

The same repeated number may be chosen from candidates **unlimited number** of times.

```

1 Note:
2
3     All numbers (including target) will be positive integers.
4     The solution set must not contain duplicate combinations.
5
6 Example 1:
7
8 Input: candidates = [2,3,6,7], target = 7,
9 A solution set is:
10 [
11     [7],
12     [2,2,3]
13 ]
14
15 Example 2:
16
17 Input: candidates = [2,3,5], target = 8,
18 A solution set is:
19 [
20     [2,2,2,2],
21     [2,3,3],
22     [3,5]
23 ]
```

DFS Backtracking. Analysis: This is still a typical combination problem, the only thing is the return level is when the sum of the path we gained is larger than the target, and we only collect the answer when it is equal. And Because a number can be used unlimited times, so that each time after we used one number, we do not increase the next start position.

```

1 def combinationSum(self, candidates, target):
2     """
3         :type candidates: List[int]
4         :type target: int
5         :rtype: List[List[int]]
6     """
7     ans = []
8     candidates.sort()
9     self.combine(candidates, target, 0, [], ans)
10    return ans
11
12 def combine(self, nums, target, s, curr, ans):
13     if target < 0:
14         return # backtracking
15     if target == 0:
16         ans.append(curr)
17         return
```

```

18     for i in range(s, len(nums)):
19         # if nums[i] > target:
20         #     return
21         self.combine(nums, target-nums[i], i, curr+[nums[i]], ans) # use i, instead of i+1 because we can reuse

```

40. Combination Sum II

Given a collection of candidate numbers (**candidates with duplicates**) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.

Each number in candidates may only **be used once** in the combination.

```

1 Note :
2
3     All numbers (including target) will be positive integers .
4     The solution set must not contain duplicate combinations .
5
6 Example 1:
7
8 Input: candidates = [10,1,2,7,6,1,5], target = 8,
9 A solution set is :
10 [
11     [1, 7],
12     [1, 2, 5],
13     [2, 6],
14     [1, 1, 6]
15 ]
16
17 Example 2:
18
19 Input: candidates = [2,5,2,1,2], target = 5,
20 A solution set is :
21 [
22     [1,2,2],
23     [5]
24 ]

```

Backtracking+Counter. Because for the first example, if we reuse the code from the previous problem, we will get extra combinations: [7, 1], [2, 1, 5]. To avoid this, we need a dictionary to save all the unique candidates with its corresponding appearing times. For a certain number, it will be used at most its counter times.

```

1 def combinationSum2(self, candidates, target):
2     """
3         :type candidates: List[int]
4         :type target: int
5         :rtype: List[List[int]]
6     """
7
8     candidates = collections.Counter(candidates)
9     ans = []
10    self.combine(list(candidates.items()), target, 0, [], ans) # convert the Counter to a list of (key, item) tuple

```

```

11     return ans
12
13 def combine(self, nums, target, s, curr, ans):
14     if target < 0:
15         return
16     if target == 0:
17         ans.append(curr)
18         return
19     for idx in range(s, len(nums)):
20         num, count = nums[idx]
21         for c in range(count):
22             self.combine(nums, target - num * (c + 1), idx + 1, curr + [num] * (c + 1), ans)

```

377. Combination Sum IV (medium)

```

1 Given an integer array with all positive numbers and no
2 duplicates, find the number of possible combinations that add
3 up to a positive integer target.
4
5 Example:
6
7 nums = [1, 2, 3]
8 target = 4
9
10 The possible combination ways are:
11 (1, 1, 1)
12 (1, 1, 2)
13 (1, 2, 1)
14 (1, 3)
15 (2, 1, 1)
16 (2, 2)
17 (3, 1)
18 Note that different sequences are counted as different
19 combinations.
20
21 Therefore the output is 7.
22
23 Follow up:
24 What if negative numbers are allowed in the given array?
25 How does it change the problem?
26 What limitation we need to add to the question to allow negative
27 numbers?

```

DFS + MEMO. This problem is similar to 39. Combination Sum. For [2, 3, 5], target = 8, comparison:

```

1 [2, 3, 5], target = 8
2 39. Combination Sum. # there is ordering (each time the start
3 index is same or larger than before)
4 [
5   [2, 2, 2, 2],
6   [2, 3, 3],

```

```

7 ]
8 377. Combination Sum IV, here we have no ordering( each time the
9   start index is the same as before). Try all element.
10 [
11   [2 ,2 ,2 ,2] ,
12   [2 ,3 ,3] ,
13   * [3 ,3 ,2]
14   * [3 ,2 ,3]
15   [3 ,5] ,
16   * [5 ,3]
17 ]

```

```

1 def combinationSum4(self ,  nums,  target):
2     """
3         :type nums: List[int]
4         :type target: int
5         :rtype: int
6     """
7     nums.sort()
8     n = len(nums)
9     def DFS(idx ,  memo,  t):
10        if t < 0:
11            return 0
12        if t == 0:
13            return 1
14        count = 0
15        if t not in memo:
16            for i in range(idx ,  n):
17                count += DFS(idx ,  memo,  t-nums[i])
18            memo[t] = count
19        return memo[t]
20    return(DFS(0 ,  {}),  target))

```

Because, here we does not need to numerate all the possible solutions, we can use dynamic programming, which will be shown in Section ??.

24.3.3 K Sum

In this subsection, we still trying to get subset that sum up to a target. But the length here is fixed. We would have 2, 3, 4 sums normally. Because it is still a combination problem, we can use the **backtracking** to do. Second, because the fixed length, we can use **multiple pointers** to build up the potential same lengthed subset. But in some cases, because the length is fixed, we can use **hashmap** to simplify the complexity.

1. Two Sum Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have **exactly** one solution, and you may not use the same element twice.

```

1 Example:
2
3 Given nums = [2 , 7 , 11 , 15] , target = 9 ,

```

```

4 Because nums[0] + nums[1] = 2 + 7 = 9,
5
6 return [0, 1].

```

Hashmap. Using backtracking or brute force will get us $O(n^2)$ time complexity. We can use hashmap to save the nums in a dictionary. Then we just check target-num in the dictionary. We would get $O(n)$ time complexity. We have two-pass hashmap and one-pass hashmap.

```

1 # two-pass hashmap
2 def twoSum(self, nums, target):
3     """
4         :type nums: List[int]
5         :type target: int
6         :rtype: List[int]
7     """
8     dict = collections.defaultdict(int)
9     for i, t in enumerate(nums):
10        dict[t] = i
11    for i, t in enumerate(nums):
12        if target - t in dict and i != dict[target-t]:
13            return [i, dict[target-t]]
14
# one-pass hashmap
15 def twoSum(self, nums, target):
16     """
17         :type nums: List[int]
18         :type target: int
19         :rtype: List[int]
20     """
21     dict = collections.defaultdict(int)
22     for i, t in enumerate(nums):
23         if target - t in dict:
24             return [dict[target-t], i]
25         dict[t] = i

```

15. 3Sum

Given an array S of n integers, are there elements a, b, c in S such that $a + b + c = 0$? Find all unique triplets in the array which gives the sum of zero.

Note: The solution set must not contain duplicate triplets.

For example, given array S = [-1, 0, 1, 2, -1, -4],

```

1 A solution set is:
2 [
3     [-1, 0, 1],
4     [-1, -1, 2]
5 ]

```

Solution: Should use three pointers, no extra space. i is the start point from [0,len-2], l,r is the other two pointers. l=i+1, r=len-1 at the beginning. The saving of time complexity is totally from the sorting algorithm.

```

1 [-4, -1, -1, 0, 1, 2]
2 i, l-> ``````<-r

```

How to delete repeat?

```

1 def threeSum(self, nums):
2     res = []
3     nums.sort()
4     for i in xrange(len(nums)-2):
5         if i > 0 and nums[i] == nums[i-1]: #make sure pointer
not repeat
6             continue
7         l, r = i+1, len(nums)-1
8         while l < r:
9             s = nums[i] + nums[l] + nums[r]
10            if s < 0:
11                l +=1
12            elif s > 0:
13                r -= 1
14            else:
15                res.append((nums[i], nums[l], nums[r]))
16                l+=1
17                r-=1
18
#after the first run, then check duplicate
example.
19
20         while l < r and nums[l] == nums[l-1]:
21             l += 1
22         while l < r and nums[r] == nums[r+1]:
23             r -= 1
24
return res

```

Use hashmap:

```

1 def threeSum(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: List[List[int]]
5     """
6     res = []
7     nums=sorted(nums)
8     if not nums:
9         return []
10    if nums[-1]<0 or nums[0]>0:
11        return []
12    end_position = len(nums)-2
13    dic_nums={}
14    for i in xrange(1,len(nums)):
15        dic_nums[nums[i]]=i# same result save the last index
16
17    for i in xrange(end_position):
18        target = 0-nums[i]
19        if i>0 and nums[i] == nums[i-1]: #this is to avoid
repeat
20            continue
21        if target<nums[i]: #if the target is smaller than
this, we can not find them on the right side
22            break

```

```

23         for j in range(i+1, len(nums)): #this is to avoid
repeat
24             if j>i+1 and nums[j]==nums[j-1]:
25                 continue
26             complement = target - nums[j]
27             if complement<nums[j]: #if the left numbers are
bigger than the complement, no need to keep searching
28                 break
29             if complement in dic_nums and dic_nums[
complement]>j: #need to make sure the complement is bigger
than nums[j]
30                 res.append([nums[i], nums[j], complement])
31
return res

```

The following code uses more time

```

1 for i in xrange(len(nums)-2):
2     if i > 0 and nums[i] == nums[i-1]:
3         continue
4     l, r = i+1, len(nums)-1
5     while l < r:
6         if l-1>=i+1 and nums[1] == nums[l-1]: #check the
front
7             l += 1
8             continue
9         if r+1<len(nums) and nums[r] == nums[r+1]:
10            r -= 1
11            continue
12         s = nums[i] + nums[l] + nums[r]
13         if s < 0:
14             l += 1
15         elif s > 0:
16             r -= 1
17         else:
18             res.append((nums[i], nums[l], nums[r]))
19             l += 1; r -= 1
20
return res

```

18. 4Sum

```

1 def fourSum(self, nums, target):
2     def findNsum(nums, target, N, result, results):
3         if len(nums) < N or N < 2 or target < nums[0]*N or
target > nums[-1]*N: # early termination
4             return
5         if N == 2: # two pointers solve sorted 2-sum problem
6             l, r = 0, len(nums)-1
7             while l < r:
8                 s = nums[l] + nums[r]
9                 if s == target:
10                     results.append(result + [nums[l], nums[r]])
11                     l += 1
12                     while l < r and nums[l] == nums[l-1]:
13                         l += 1

```

```

14             l += 1
15         while l < r and nums[r] == nums[r+1]:
16             r -= 1
17     elif s < target:
18         l += 1
19     else:
20         r -= 1
21     else: # recursively reduce N
22         for i in range(len(nums)-N+1):
23             if i == 0 or (i > 0 and nums[i-1] != nums[i]):
24                 findNsum(nums[i+1:], target-nums[i], N-1,
25                           result+[nums[i]], results) #reduce nums size, reduce
26                           target, save result
27
28 results = []
29 findNsum(sorted(nums), target, 4, [], results)
30 return results

```

454. 4Sum II

Given four lists A, B, C, D of integer values, compute how many tuples (i, j, k, l) there are such that $A[i] + B[j] + C[k] + D[l]$ is zero.

To make problem a bit easier, all A, B, C, D have same length of N where $0 \leq N \leq 500$. All integers are in the range of -228 to 228–1 and the result is guaranteed to be at most 231–1.

Example:

```

1 Input:
2 A = [ 1,  2]
3 B = [-2,-1]
4 C = [-1,  2]
5 D = [ 0,  2]
6
7 Output:
8 2

```

Explanation:

```

1 The two tuples are:
2 1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) +
   2 = 0
3 2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) +
   0 = 0

```

Solution: if we use brute force, use 4 for loop, then it is $O(N^4)$. If we use divide and conquer, sum the first half, and save a dictionary (counter), time complexity is $O(2N^2)$. What if we have 6 sum, we can reduce it to $O(2N^3)$, what if 8 sum.

```

1 def fourSumCount(self, A, B, C, D):
2     AB = collections.Counter(a+b for a in A for b in B)
3     return sum(AB[-c-d] for c in C for d in D)

```

Summary

As we have seen from the shown examples in this section, to solve the combination problem, backtrack shown in Section ?? offers a universal solution. Also, there is another iterative solution which suits the power set purpose. And I would include its code here again:

```

1 def subsets(self, nums):
2     result = [[]] #use two dimensional, which already have []
3     one_element
4     for num in nums:
5         new_results = []
6         for r in result:
7             new_results.append(r + [num])
8         result += new_results
9
10 return result

```

If we have duplicates, how to handle in the backtrack?? In the iterative solution, we can replace the array with a dictionary saves the counts.

24.3.4 Permutation

46. Permutations

```

1 Given a collection of distinct numbers, return all possible
2   permutations.
3
4 For example,
5   [1,2,3] have the following permutations:
6
7 [
8   [1,2,3],
9   [1,3,2],
10  [2,1,3],
11  [2,3,1],
12  [3,1,2],
13  [3,2,1]
14 ]

```

47. Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,

```

1 [1,1,2] have the following unique permutations:
2
3 [
4   [1,1,2],
5   [1,2,1],
6   [2,1,1]
7 ]

```

301. Remove Invalid Parentheses

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses (and).

Examples:

```

1 "()()()" -> ["()", "((())()"]
2 "(a)()()" -> ["(a)()", "(a())()"]
3 ")"(" -> []

```

24.4 Merge and Partition

24.4.1 Merge Lists

We can use divide and conquer (see the merge sort) and the priority queue.

24.4.2 Partition Lists

Partition of lists can be converted to subarray, combination, subsequence problems. For example,

1. 416. Partition Equal Subset Sum (combination)
2. 698. Partition to K Equal Sum Subsets

24.5 Intervals

Sweep Line is a type of algorithm that mainly used to solve problems with intervals of one-dimensional. Let us look at one example: 1. 253. Meeting Rooms II

Given an array of meeting time intervals consisting of start and end times $[[s_1, e_1], [s_2, e_2], \dots]$ ($s_i < e_i$), find the minimum number of conference rooms required.

```

1 Example 1:
2
3 Input: [[0, 30], [5, 10], [15, 20]]
4 Output: 2
5
6 Example 2:
7
8 Input: [[7, 10], [2, 4]]
9 Output: 1

```

It would help a lot if at first we can draw one example with coordinates. First, the simplest situation is when we only need one meeting room is there is no intersection between these time intervals. If we add one interval that only intersect with one of the previous intervals, this means we need



Figure 24.2: Interval questions

two conference rooms. So to find the minimum conference rooms we need, we need to find the maximum number of intersection between these time intervals. The most native solution is to scan all the time slot in one for loop, and at another inner loop go through all the intervals, if this time slot is in this intervals, then we increase the minimum number of meeting room counter. This gives us time complexity of $O(n * m)$, where n is the number of intervals and m is the total number of time slots. The Python code is as follows, unfortunately, with this solution we have LTE error.

```

1 # Definition for an interval.
2 # class Interval(object):
3 #     def __init__(self, s=0, e=0):
4 #         self.start = s
5 #         self.end = e
6
7 from collections import defaultdict
8 from heapq import heappush, heappop
9 from sys import maxint
10 class Solution(object):
11     def minMeetingRooms(self, intervals):
12         """
13             :type intervals: List[Interval]
14             :rtype: int
15         """
16         if not intervals:
17             return 0
18         #solution 1, voting, time complexity is O(e1-s1), 71/77
19         test, TLE
20         votes = defaultdict(int)
21         num_rooms = 0
22         for interval in intervals:
23             s=interval.start
24             e=interval.end

```

```

24     for i in range(s+1,e+1):
25         votes[i] += 1
26         num_rooms = max(num_rooms, votes[i])
27     return num_rooms

```

24.5.1 Speedup with Sweep Line

Now, let us see how to speed up this process. We can use Sweep Line method. For the sweep line, we have three basic implementations: one-dimensional, min-heap, or map based.

One-dimensional Implementation

To get the maximum number of intersection of all the intervals, it is not necessarily to scan all the time slots, how about just scan the key slot: the starts and ends. Thus, what we can do is to open an array and put all the start or end slot into the array, and with 1 to mark it as start and 0 to mark it as end. Then we sort this array. Till this point, how to get the maximum intersection? We go through this sorted array, if we get a start our current number of room needed will increase by one, otherwise, if we encounter an end slot, it means one meeting room is freed, thus we decrease the current on-going meeting room by one. We use another global variable to track the maximum number of rooms needed in this whole process. Great, because now our time complexity is decided by the number of slots $2n$, with the sorting algorithm, which makes the whole time complexity $O(n \log n)$ and space complexity n . This speeded up algorithm is called Sweep Line algorithm. Before we write our code, we better check the *special cases*, what if there is one slot that is marked as start in one interval but is the end of another interval. This means we can not increase the counting at first, but we need to decrease, so that the sorting should be based on the first element of the tuple, and followed by the second element of the tuple. For example, the simple case $[[13, 15], [1, 13]]$, we only need maximum of one meeting room. Thus it can be implemented as:

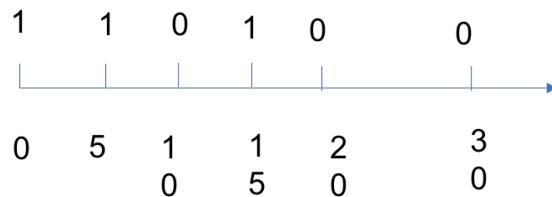


Figure 24.3: One-dimensional Sweep Line

```

1 def minMeetingRooms(self, intervals):
2     if not intervals:

```

```

3         return 0
4 #solution 2
5 slots = []
6 # put slots into one-dimensional axis
7 for i in intervals:
8     slots.append((i.start, 1))
9     slots.append((i.end, 0))
10 # sort these slots on this dimension
11 #slots.sort(key = lambda x: (x[0], x[1]))
12 slots.sort()
13
14 # now execute the counting
15 crt_room, max_room = 0, 0
16 for s in slots:
17     if s[1]==0: # if it ends, decrease
18         crt_room-=1
19     else:
20         crt_room+=1
21     max_room = max(max_room, crt_room)
22 return max_room

```

Min-heap Implementation

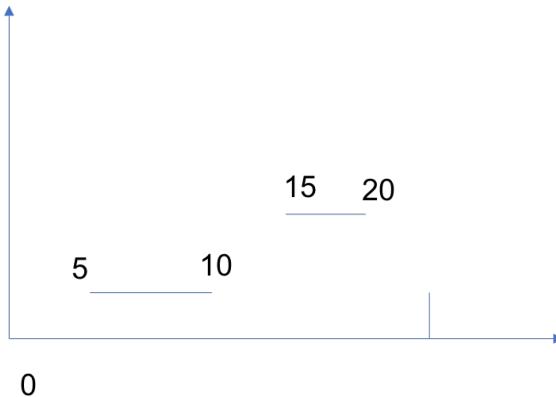


Figure 24.4: Min-heap for Sweep Line

Instead of opening an array to save all the time slots, we can directly sort the intervals in the order of the start time. We can see Fig. 24.4, we go through the intervals and visit their end time, the first one we encounter is 30, we put it in a min-heap, and then we visit the next interval [5, 10], 5 is smaller than the previous end time 30, it means this interval intersected with a previous interval, so the number of maximum rooms increase 1, we get 2 rooms now. We put 10 into the min-heap. Next, we visit [15, 20], 15 is larger than the first element in the min-heap 10, it means that these two intervals can be merged into one [5, 20], so we need to update the end time

10 to 20.

This way, the time complexity is still the same which is decided by the sorting algorithm. While the space complexity is decided by real situation, it varies from $O(1)$ (no intersection) to $O(n)$ (all the meetings are intersected at least one time slot).

```

1 def minMeetingRooms(self, intervals):
2     if not intervals:
3         return 0
4     #solution 2
5     intervals.sort(key=lambda x:x.start)
6     h = [intervals[0].end]
7     rooms = 1
8     for i in intervals[1:]:
9         s,e=i.start, i.end
10        e_before = h[0]
11        if s<e_before: #overlap
12            heappush(h, i.end)
13            rooms+=1
14        else: #no overlap
15            #merge
16            heappop(h) #kick out 10 in our example
17            heappush(h,e) # replace 10 with 20
18    return rooms

```

Map-based Implementation

```

1 class Solution {
2 public:
3     int minMeetingRooms(vector<Interval>& intervals) {
4         map<int, int> mp;
5         for (auto val : intervals) {
6             ++mp[val.start];
7             --mp[val.end];
8         }
9         int max_room = 0, crt_room = 0;
10        for (auto val : mp) {
11            crt_room += val.second;
12            max_room = max(max_room, crt_room);
13        }
14        return max_room;
15    }
16 };

```

24.5.2 LeetCode Problems

- 986. Interval List Intersections** Given two lists of closed intervals, each list of intervals is pairwise disjoint and in sorted order. Return the intersection of these two interval lists.

```

Input: A = [[0,2],[5,10],[13,23],[24,25]], B =
[[1,5],[8,12],[15,24],[25,26]]
Output: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
Reminder: The inputs and the desired output are lists of
Interval objects, and not arrays or lists.

```

24.6 Intersection

For problems to get intersections of lists, we can use hashmap, which takes $O(m + n)$ time complexity. Also, we can use sorting at first and use two pointers one start from the start of each array. Examples are shown as below;

1. 349. Intersection of Two Arrays (Easy)

Given two arrays, write a function to compute their intersection.

Example:

```
1 Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2].
```

Note:

- Each element in the result must be unique.
- The result can be in any order.

Solution 1: Using hashmap, here we use set to convert, this takes 43ms.

```

1 def intersection(self, nums1, nums2):
2     """
3     :type nums1: List[int]
4     :type nums2: List[int]
5     :rtype: List[int]
6     """
7     if not nums1 or not nums2:
8         return []
9     if len(nums1) > len(nums2):
10        nums1, nums2 = nums2, nums1
11    ans = set()
12    nums1 = set(nums1)
13    for e in nums2:
14        if e in nums1:
15            ans.add(e)
16    return list(ans)

```

Solution2: sorting at first, and then use pointers. Take 46 ms.

```

1 def intersection(self, nums1, nums2):
2     """
3     :type nums1: List[int]
4     :type nums2: List[int]

```

```

5   :rtype: List[int]
6
7   """
8   nums1.sort()
9   nums2.sort()
10  r = set()
11  i, j = 0, 0
12  while i < len(nums1) and j < len(nums2):
13      if nums1[i] < nums2[j]:
14          i += 1
15      elif nums1[i] > nums2[j]:
16          j += 1
17      else:
18          r.add(nums1[i])
19          i += 1
20  return list(r)

```

2. 350. Intersection of Two Arrays II(Easy)

Given two arrays, write a function to compute their intersection.

Example:

```
1 Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2, 2].
```

Note:

- Each element in the result should appear as many times as it shows in both arrays.
- The result can be in any order.

Follow up:

- What if the given array is already sorted? How would you optimize your algorithm?
- What if nums1's size is small compared to nums2's size? Which algorithm is better?
- What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

24.7 Miscellaneous Questions

24.13 283. Move Zeroes. (Easy) Given an array `nums`, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Note:

1. You must do this in-place without making a copy of the array.

2. Minimize the total number of operations.

```

1 Example :
2
3 Input : [0 ,1 ,0 ,3 ,12]
4 Output: [1 ,3 ,12 ,0 ,0]
```

Solution 1: Find All Zeros Subarray. If we found the first all zeros subarray $[0, \dots, 0] + [x]$, and we can swap this subarray with the first non-zero element as swap last 0 with x, swap second last element with x, ..., and so on. Therefore, if 0 is at first index, one zero, then it takes $O(n)$, if another 0, at index 1, it takes $n-1+n-2 = 2n$. It is bit tricky to compute the complexity analysis. The upper bound is $O(n^2)$.

24.8 Exercises

24.8.1 Subsequence with (DP)

594. Longest Harmonious Subsequence

We define a harmonious array is an array where the difference between its maximum value and its minimum value is exactly 1.

Now, given an integer array, you need to find the length of its longest harmonious subsequence among all its possible subsequences.

Example 1:

```

1 Input : [1 ,3 ,2 ,2 ,5 ,2 ,3 ,7]
2 Output: 5
3 Explanation: The longest harmonious subsequence is
   [3 ,2 ,2 ,2 ,3].
```

Note: The length of the input array will not exceed 20,000.

Solution: at first, use a Counter to save the whole set. Then visit the counter dictionary, to check key+1 and key-1, only when the item is not zero, we can count it as validate, or else it is 0.

```

1 from collections import Counter
2 class Solution:
3     def findLHS(self , nums):
4         """
5             :type nums: List[int]
6             :rtype: int
7         """
8         if not nums or len(nums)<2:
9             return 0
10        count=Counter(nums) #the list is sorted by the key
11        value
12        maxLen = 0
```

```

12     for key, item in count.items(): #to visit the key:
13         item in the counter
14         if count[key+1]: #because the list is sorted,
15             so we only need to check key+1
16             maxLen = max(maxLen, item+count[key+1])
17
18     # if count[key-1]:
19     #     maxLen=max(maxLen, item+count[key-1])
20
21     return maxLen

```

2. 521. Longest Uncommon Subsequence I

Given a group of two strings, you need to find the longest uncommon subsequence of this group of two strings. The longest uncommon subsequence is defined as the longest subsequence of one of these strings and this subsequence should not be any subsequence of the other strings.

A subsequence is a sequence that can be derived from one sequence by deleting some characters without changing the order of the remaining elements. Trivially, any string is a subsequence of itself and an empty string is a subsequence of any string.

The input will be two strings, and the output needs to be the length of the longest uncommon subsequence. If the longest uncommon subsequence doesn't exist, return -1.

Example 1:

```

1 Input: "aba", "cdc"
2 Output: 3
3 Explanation: The longest uncommon subsequence is "aba" (or
               "cdc"),
4 because "aba" is a subsequence of "aba",
5 but not a subsequence of any other strings in the group of
               two strings.

```

Note:

Both strings' lengths will not exceed 100.

Only letters from a z will appear in input strings.

Solution: if we get more examples, we could found the following rules, “aba”, “aba” return -1,

```

1 def findLUSlength(self, a, b):
2     """
3         :type a: str
4         :type b: str
5         :rtype: int
6     """
7     if len(b) != len(a):
8         return max(len(a), len(b))

```

```

9     #length is the same
10    return len(a) if a!=b else -1

```

3. 424. Longest Repeating Character Replacement

Given a string that consists of only uppercase English letters, you can replace any letter in the string with another letter at most k times. Find the length of a longest substring containing all repeating letters you can get after performing the above operations.

Note:

Both the string's length and k will not exceed 104.

Example 1:

```

1 Input :
2 s = "ABAB" , k = 2
3
4 Output :
5 4

```

Explanation: Replace the two 'A's with two 'B's or vice versa.

Example 2:

```

1 Input :
2 s = "AABABBA" , k = 1
3
4 Output :
5 4

```

Explanation: Replace the one 'A' in the middle with 'B' and form "AABBBA". The substring "BBBB" has the longest repeating letters, which is 4.

Solution: the brute-force recursive solution for this, is try to replace any char into another when it is not equal or choose not too. LTE

```

1 #brute force , use recursive function to write brute force
2     solution
3         def replace(news , idx , re_char , k):
4             nonlocal maxLen
5             if k==0 or idx==len(s):
6                 maxLen = max(maxLen , getLen(news))
7                 return
8
9         if s[idx]!=re_char: #replace
10            news_copy=news[:idx]+re_char+news[idx+1:]
11            replace(news_copy , idx+1, re_char , k-1)
12            replace(news[:] , idx+1, re_char , k)
13
14            #what if we only have one char
15            # for char1 in chars.keys():
16            #     replace(s[:] , 0 , char1 , k)

```

To get the BCR, think about the sliding window. The longest repeating string we can by number of replacement = ‘length of string max(number of occurrence of letter i), i=’A’ to ‘Z’. With the constraint, which means the equation needs to be $\leq k$. So we can use sliding window to record the max occurrence, and when the constraint is violated, we shrink the window. Given an example, strs= “BBCABB-BAB”, k=2, when i=0, and j=7, $8-5=3>2$, which is at A, we need to shrink it, the maxCharCount changed to 4, i=1, so that $8-1-4=3$, i=2, $8-2-3=3$, $8-3-3=2$, so i=3, current length is 5.

```

1 def characterReplacement(self, s, k):
2     """
3     :type s: str
4     :type k: int
5     :rtype: int
6     """
7     i, j = 0, 0 #sliding window
8     counter = [0] * 26
9     ans = 0
10    maxCharCount = 0
11    while j < len(s):
12        counter[ord(s[j]) - ord('A')] += 1
13        maxCharCount = max(maxCharCount, counter[ord(s[
14            j]) - ord('A')])
15        while j - i + 1 - maxCharCount > k: #now shrink the
16            window
17                counter[ord(s[i]) - ord('A')] -= 1
18                i += 1
19                #update max
20                maxCharCount = max(counter)
21                ans = max(ans, j - i + 1)
22                j += 1
23
24    return ans

```

4. 395. Longest Substring with At Least K Repeating Characters

Find the length of the longest substring T of a given string (consists of lowercase letters only) such that every character in T appears no less than k times.

Example 1:

```

1 Input:
2 s = "aaabb", k = 3
3
4 Output:
5 3

```

The longest substring is "aaa", as 'a' is repeated 3 times.

Example 2:

```

1 Input :
2 s = "ababbc" , k = 2
3
4 Output :
5 5

```

The longest substring is "ababb", as 'a' is repeated 2 times and 'b' is repeated 3 times.

Solution: use dynamic programming with memo: Cons: it takes too much space, and with LTE.

```

1 from collections import Counter, defaultdict
2 class Solution:
3     def longestSubstring(self, s, k):
4         """
5             :type s: str
6             :type k: int
7             :rtype: int
8         """
9         if not s:
10             return 0
11         if len(s)<k:
12             return 0
13         count = Counter(char for char in s)
14         print(count)
15         memo=[[None for col in range(len(s))] for row in
16               range(len(s))]
17
18     def cut(start,end,count):
19         if start>end:
20             return 0
21         if memo[start][end]==None:
22             if any(0<item<k for key,item in count.items()):
23                 newCounterF=count.copy()
24                 newCounterF[s[start]]-=1
25                 newCounterB=count.copy()
26                 newCounterB[s[end]]-=1
27                 #print(newsF,newsB)
28                 memo[start][end]= max(cut(start+1, end,
29                     newCounterF), cut(start, end-1, newCounterB))
30             else:
31                 memo[start][end] = end-start+1
32         return memo[start][end]
33     return cut(0,len(s)-1,count)

```

Now, use sliding window, we use a pointer mid, what start from 0, if the whole string satisfy the condition, return len(s). Otherwise, use two while loop to separate the string into three substrings: left, mid, right. left satisfy, mid unsatisfy, right unknown.

```

1 from collections import Counter, defaultdict
2 class Solution:

```

```

3     def longestSubstring(self, s, k):
4         """
5             :type s: str
6             :type k: int
7             :rtype: int
8         """
9         if not s:
10             return 0
11         if len(s) < k:
12             return 0
13         count = Counter(char for char in s)
14         mid=0 #on the left side , from 0-mid, satisfied
15         elements
16         while mid<len(s) and count[s[mid]]>=k:
17             mid+=1
18         if mid==len(s): return len(s)
19         left = self.longestSubstring(s[:mid],k) # "ababb"
20         #from pre_mid - cur_mid, get rid of those can't
21         # satisfy the condition
22         while mid<len(s) and count[s[mid]]<k:
23             mid+=1
24         #now the right side keep doing it
25         right = self.longestSubstring(s[mid:],k)
26         return max(left,right)

```

24.8.2 Subset

216. Combination Sum III

Find all possible combinations of **k numbers** that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Note :

All numbers will be positive integers.
The solution set must not contain duplicate combinations.

Example 1:

Input: k = 3, n = 7
Output: [[1, 2, 4]]

Example 2:

Input: k = 3, n = 9
Output: [[1, 2, 6], [1, 3, 5], [2, 3, 4]]

```

1 def combinationSum3(self, k, n):
2     """
3         :type k: int
4         :type n: int
5         :rtype: List[List[int]]
6     """

```

```

7      # each only used one time
8      def combine(s, curr, ans, t, d, k, n):
9          if t < 0:
10             return
11         if d == k:
12             if t == 0:
13                 ans.append(curr)
14             return
15         for i in range(s, n):
16             num = i+1
17             combine(i+1, curr+[num], ans, t-num, d+1, k, n)
18         ans = []
19         combine(0, [], ans, n, 0, k, 9)
20     return ans

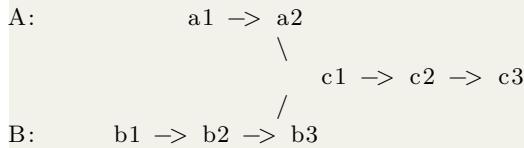
```

24.8.3 Intersection

160. Intersection of Two Linked Lists (Easy)

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:



begin to intersect at node c1.

Notes:

- If the two linked lists have no intersection at all, return null.
- The linked lists must retain their original structure after the function returns.
- You may assume there are no cycles anywhere in the entire linked structure.
- Your code should preferably run in $O(n)$ time and use only $O(1)$ memory.

25

Linked List, Stack, Queue, and Heap Questions (12%)

In this chapter, we focusing on solving problems that carried on or the solution is related using non-linear data structures that are not array/string, such as linked list, heap, queue, and stack.

25.1 Linked List

Problems with linked list can be basic operations to add or remove node, or merge two different linked list.

Circular Linked List For the circular linked list, when we are traversing the list, the most important thing is to know how to set up the end condition for the while loop.

25.1 708. Insert into a Cyclic Sorted List (medium) Given a node from a cyclic linked list which is sorted in ascending order, write a function to insert a value into the list such that it remains a cyclic sorted list. The given node can be a reference to any single node in the list, and may not be necessarily the smallest value in the cyclic list. For example,

Analysis: The maximum we traverse the list is one round. The potential positions we insert is related to the insert value. Suppose the linked list is in range of $[s, e]$, $s \leq e$. Given the insert value as m :

1. $m \in [s, e]$: we insert in the middle of the list.
2. $m \geq e$ or $m \leq s$: we insert at the end of the list, we need to detect the end as if the current node's value is larger than its successor's value.

590 25. LINKED LIST, STACK, QUEUE, AND HEAP QUESTIONS (12%)

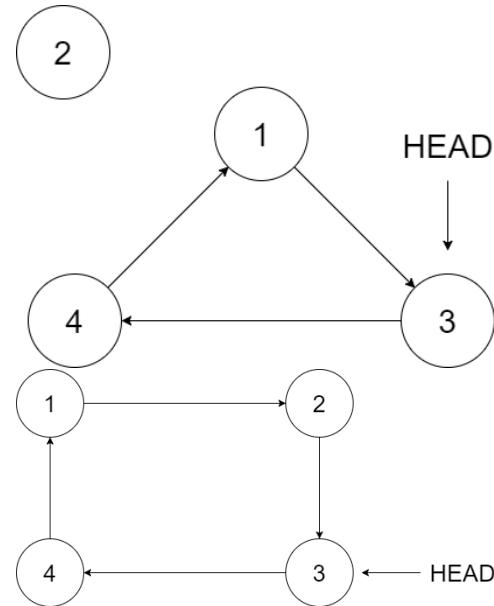


Figure 25.1: Example of insertion in circular list

3. After one loop, if we can not find a place, then we insert at the end. For example, 2->2->2 and insert 3 or 2->3->4->2 and insert 2.

```

1 def insert(self, head, insertVal):
2     if not head: # 0 node
3         head = Node(insertVal, None)
4         head.next = head
5         return head
6
7     cur = head
8     while cur.next != head:
9         if cur.val <= insertVal <= cur.next.val: # insert
10            break
11        elif cur.val > cur.next.val: # end and start
12            if insertVal >= cur.val or insertVal <= cur.
13                next.val:
14                break
15            cur = cur.next
16        else:
17            cur = cur.next
18    # insert
19    node = Node(insertVal, None)
20    node.next, cur.next = cur.next, node
21    return head
  
```

25.2 Queue and Stack

Because Queue and Stack is used to implement BFS and DFS search respectively, therefore, that type of implementation is covered in Chapter ???. The other problems include: Buffering problem with Queue(circular queue),

25.2.1 Implementing Queue and Stack

25.2 622. Design Circular Queue (medium). Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called "**Ring Buffer**".

Your implementation should support following operations:

- MyCircularQueue(k): Constructor, set the size of the queue to be k.
- Front: Get the front item from the queue. If the queue is empty, return -1.
- Rear: Get the last item from the queue. If the queue is empty, return -1.
- enQueue(value): Insert an element into the circular queue. Return true if the operation is successful.
- deQueue(): Delete an element from the circular queue. Return true if the operation is successful.
- isEmpty(): Checks whether the circular queue is empty or not.
- isFull(): Checks whether the circular queue is full or not.

Solution 1: Singly Linked List with Predefined Size. This is a typical queue data structure and because it is a buffering, therefore, we need to limit its size. As shown in previous theory chapter of the book, queue can be implemented with singly linked list with two pointers, one at the head and the other at the rear. The additional controlling we need is to limit the size of the queue.

```

1 class MyCircularQueue:
2     class Node:
3         def __init__(self, val):
4             self.val = val
5             self.next = None
6     def __init__(self, k):
7         self.size = k
8         self.head = None
9         self.tail = None
10        self.cur_size = 0
11

```

592 25. LINKED LIST, STACK, QUEUE, AND HEAP QUESTIONS (12%)

```
12     def enQueue(self, value):
13         if self.cur_size >= self.size:
14             return False
15         new_node = MyCircularQueue.Node(value)
16         if self.cur_size == 0:
17             self.tail = self.head = new_node
18         else:
19             self.tail.next = new_node
20             new_node.next = self.head
21             self.tail = new_node
22         self.cur_size += 1
23         return True
24
25     def deQueue(self):
26
27         if self.cur_size == 0:
28             return False
29         # delete head node
30         val = self.head.val
31         if self.cur_size == 1:
32             self.head = self.tail = None
33         else:
34             self.head = self.head.next
35         self.cur_size -= 1
36         return True
37
38     def Front(self):
39         return self.head.val if self.head else -1
40
41     def Rear(self):
42         return self.tail.val if self.tail else -1
43
44     def isEmpty(self):
45         return True if self.cur_size == 0 else False
46
47     def isFull(self):
48         return True if self.cur_size == self.size else False
```

25.3 641. Design Circular Deque (medium).

Solution: Doubly linked List with Predefined size

25.2.2 Solving Problems Using Queue

Use as a Buffer

25.4 346. Moving Average from Data Stream (easy). Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

Example :

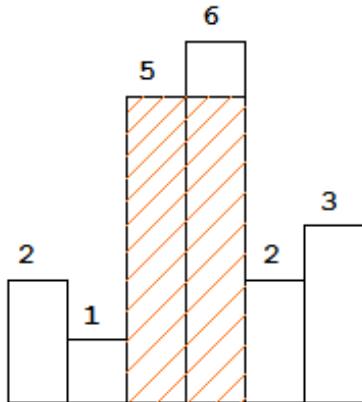


Figure 25.2: Histogram

```
MovingAverage m = new MovingAverage(3);
m.next(1) = 1
m.next(10) = (1 + 10) / 2
m.next(3) = (1 + 10 + 3) / 3
m.next(5) = (10 + 3 + 5) / 3
```

Solution: module deque with maxlen. When we have a fixed window size, this is like a buffer, it has a maximum of capacity. When the $n+1$ th element come, we need delete the leftmost element first. This is directly implemented in deque module if we set the maxlen to the size we want. Also, it is easy to use function like sum() and len() to compute the average value.

```
1 from collections import deque
2 class MovingAverage:
3     def __init__(self, size):
4         self.q = deque(maxlen = size)
5     def next(self, val):
6         self.q.append(val)
7         return sum(self.q)/len(self.q)
```

25.2.3 Solving Problems with Stack and Monotone Stack

84. Largest Rectangle in Histogram

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram. Above is a histogram where width of each bar is 1, given height = [2, 1, 5, 6, 2, 3]. The largest rectangle is shown in the shaded area, which has area = 10 unit.

Solution: brute force. Start from 2 which will be included, then we go to the right side to find the minimum height, we could have possible

594 25. LINKED LIST, STACK, QUEUE, AND HEAP QUESTIONS (12%)

area ($1 \times 2, 1 \times 3, 1 \times 4, 1 \times 5, \dots$), which gave us $O(n^2)$ to track the min height and width.

```

1 class Solution:
2     def largestRectangleArea(self, heights):
3         """
4             :type heights: List[int]
5             :rtype: int
6         """
7         if not heights:
8             return 0
9         maxsize = max(heights)
10
11        for i in range(len(heights)):
12            minheight = heights[i]
13            width = 1
14            for j in range(i+1, len(heights)):
15                width+=1
16                minheight = min(minheight, heights[j])
17            maxsize = max(maxsize, minheight*width)
18        return maxsize

```

Now, try the BCR, which is $O(n)$. The maximum area is among areas that use each height as the rectangle height multiplied by the width that works. For the above example, we would choose the maximum among $2 \times 1, 1 \times 6, 5 \times 2, 6 \times 1, 2 \times 4, 3 \times 1$. So, the important step here is to find the possible width, for element 2, if the following height is increasing, then the width grows, however, since the following height 1 is smaller, so 2 will be popped out, we can get 2×1 , which satisfies the condition of the monotonic increasing stack, when one element is popped out, which means we found the next element that is smaller than the kicked out element, so the width span ended here. How to deal if current number equals to previous, 6,6,6,6,6, we need to pop previous, and append current. The structure we use here is called Monotonic Stack, which will only allow the increasing elements to get in the stack, and once smaller or equal ones get in, it kicks out the previous smaller elements.

```

1 def largestRectangleArea(self, heights):
2     """
3         :type heights: List[int]
4         :rtype: int
5     """
6     if not heights:
7         return 0
8     maxsize = max(heights)
9
10    stack = [-1]
11
12    #the stack will only grow
13    for i, h in enumerate(heights):

```

```

14     if stack[-1]!=-1:
15         if h>heights[stack[-1]]:
16             stack.append(i)
17         else:
18             #start to kick to pop and compute the
19             area
20             while stack[-1]!=-1 and h<=heights[
21                 stack[-1]]: #same or equal needs to be pop out
22                 idx = stack.pop()
23                 v = heights[idx]
24                 maxsize=max(maxsize, (i-stack
25                   [-1]-1)*v)
26             stack.append(i)
27
28     else:
29         stack.append(i)
30 #handle the left stack
31     while stack[-1]!=-1:
32         idx = stack.pop()
33         v = heights[idx]
34         maxsize=max(maxsize, (len(heights)-stack[-1]-1)
35           *v)
36
37     return maxsize

```

85. Maximal Rectangle Solution: 64/66 with LTE

```

1 def maximalRectangle(self, matrix):
2     """
3     :type matrix: List[List[str]]
4     :rtype: int
5     """
6     if not matrix:
7         return 0
8     if len(matrix[0])==0:
9         return 0
10    row, col = len(matrix), len(matrix[0])
11
12    def check(x,y,w,h):
13        #check the last col
14        for i in range(x, x+h): #change row
15            if matrix[i][y+w-1]=='0':
16                return 0
17        for j in range(y, y+w): #change col
18            if matrix[x+h-1][j]=='0':
19                return 0
20        return w*h
21    maxsize = 0
22    for i in range(row):
23        for j in range(col): #start point i,j
24            if matrix[i][j]=='0':
25                continue
26            for h in range(1, row-i+1): #decide the
27                for w in range(1, col-j+1):

```

596 25. LINKED LIST, STACK, QUEUE, AND HEAP QUESTIONS (12%)

```

28             rslt = check(i,j,w,h)
29             if rslt==0: #we definitely need to
30                 break it. or else we get wrong result
31             break
32             maxsize = max(maxsize, check(i,j,w,
h))
32     return maxsize

```

Now, the same as before, use the sums

```

1 def maximalRectangle(self, matrix):
2     """
3     :type matrix: List[List[str]]
4     :rtype: int
5     """
6     if not matrix:
7         return 0
8     if len(matrix[0]) == 0:
9         return 0
10    row, col = len(matrix), len(matrix[0])
11    sums = [[0 for _ in range(col+1)] for _ in range(
row+1)]
12    #no need to initialize row 0 and col 0, because we
just need it to be 0
13    for i in range(1, row+1):
14        for j in range(1, col+1):
15            sums[i][j] = sums[i-1][j] + sums[i][j-1] - sums[i-
1][j-1] + [0, 1][matrix[i-1][j-1] == '1']
16
17    def check(x, y, w, h):
18        count = sums[x+h-1][y+w-1] - sums[x+h-1][y-1] -
sums[x-1][y+w-1] + sums[x-1][y-1]
19        return count if count == w*h else 0
20
21 maxsize = 0
22 for i in range(row):
23     for j in range(col): #start point i, j
24         if matrix[i][j] == '0':
25             continue
26         for h in range(1, row-i+1): #decide the
size of the window
27             for w in range(1, col-j+1):
28                 rslt = check(i+1, j+1, w, h)
29                 if rslt == 0: #we definitely need to
break it. or else we get wrong result
30                     break
31                 maxsize = max(maxsize, rslt)
32     return maxsize

```

Still can not be AC. So we need another solution. Now use the largest rectangle in histogram.

```

1 def maximalRectangle(self, matrix):
2     """

```

```

3     :type matrix: List[List[str]]
4     :rtype: int
5     """
6     if not matrix:
7         return 0
8     if len(matrix[0]) == 0:
9         return 0
10    def getMaxAreaHist(heights):
11        if not heights:
12            return 0
13        maxsize = max(heights)
14
15    stack = [-1]
16
17    #the stack will only grow
18    for i, h in enumerate(heights):
19        if stack[-1] != -1:
20            if h > heights[stack[-1]]:
21                stack.append(i)
22            else:
23                #start to kick to pop and compute
24                the area
25                while stack[-1] != -1 and h <= heights[
26                    stack[-1]]: #same or equal needs to be pop out
27                    idx = stack.pop()
28                    v = heights[idx]
29                    maxsize = max(maxsize, (i - stack
30                    [-1] - 1) * v)
31                    stack.append(i)
32
33    else:
34        stack.append(i)
35        #handle the left stack
36        while stack[-1] != -1:
37            idx = stack.pop()
38            v = heights[idx]
39            maxsize = max(maxsize, (len(heights) - stack
40            [-1] - 1) * v)
41        return maxsize
42    row, col = len(matrix), len(matrix[0])
43    heights = [0] * col #save the maximum heights till
44    here
45    maxsize = 0
46    for r in range(row):
47        for c in range(col):
48            if matrix[r][c] == '1':
49                heights[c] += 1
50            else:
51                heights[c] = 0
52            #print(heights)
53            maxsize = max(maxsize, getMaxAreaHist(heights))
54    return maxsize

```

Monotonic Stack

598 25. LINKED LIST, STACK, QUEUE, AND HEAP QUESTIONS (12%)

122. Best Time to Buy and Sell Stock II

Say you have an array for which the i th element is the price of a given stock on day i .

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

```
1 Input: [7,1,5,3,6,4]
2 Output: 7
3 Explanation: Buy on day 2 (price = 1) and sell on day 3 (
    price = 5), profit = 5-1 = 4.
4             Then buy on day 4 (price = 3) and sell on day
5             5 (price = 6), profit = 6-3 = 3.
```

Example 2:

```
1 Input: [1,2,3,4,5]
2 Output: 4
3 Explanation: Buy on day 1 (price = 1) and sell on day 5 (
    price = 5), profit = 5-1 = 4.
4             Note that you cannot buy on day 1, buy on day
5             2 and sell them later, as you are
             engaging multiple transactions at the same
             time. You must sell before buying again.
```

Example 3:

```
1 Input: [7,6,4,3,1]
2 Output: 0
3 Explanation: In this case, no transaction is done, i.e. max
    profit = 0.
```

Solution: the difference compared with the first problem is that we can have multiple transaction, so whenever we can make profit we can have an transaction. We can notice that if we have [1,2,3,5], we only need one transaction to buy at 1 and sell at 5, which makes profit 4. This problem can be resolved with decreasing monotonic stack. whenever the stack is increasing, we kick out that number, which is the smallest number so far before i and this is the transaction that make the biggest profit = current price - previous element. Or else, we keep push smaller price inside the stack.

```
1 def maxProfit(self, prices):
2     """
3     :type prices: List[int]
4     :rtype: int
5     """
```

```

6     mono_stack = []
7     profit = 0
8     for p in prices:
9         if not mono_stack:
10            mono_stack.append(p)
11        else:
12            if p<mono_stack[-1]:
13                mono_stack.append(p)
14            else:
15                #kick out till it is decreasing
16                if mono_stack and mono_stack[-1]<p:
17                    price = mono_stack.pop()
18                    profit += p-price
19
20            while mono_stack and mono_stack[-1]<p:
21                price = mono_stack.pop()
22            mono_stack.append(p)
23
24    return profit

```

Also, there are other solutions that can use $O(1)$ space. Say the given array is: [7, 1, 5, 3, 6, 4]. If we plot the numbers of the given array on a graph, we get: If we analyze the graph, we notice that the points of

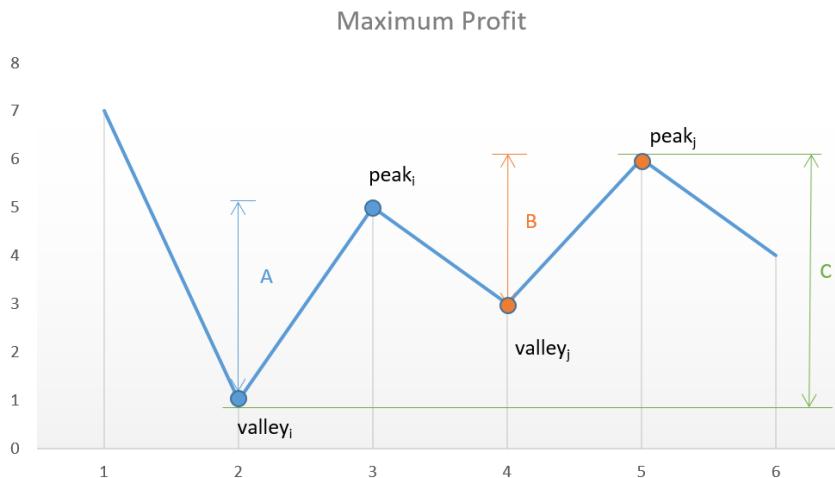


Figure 25.3: Track the peaks and valleys

interest are the consecutive valleys and peaks.

Mathematically speaking:

$$\text{TotalProfit} = \sum_i (\text{height}(\text{peak}_i) - \text{height}(\text{valley}_i)) \quad (25.1)$$

The key point is we need to consider every peak immediately following a valley to maximize the profit. In case we skip one of the peaks (trying

600 25. LINKED LIST, STACK, QUEUE, AND HEAP QUESTIONS (12%)

to obtain more profit), we will end up losing the profit over one of the transactions leading to an overall lesser profit.

```

1 class Solution {
2     public int maxProfit(int[] prices) {
3         int i = 0;
4         int valley = prices[0];
5         int peak = prices[0];
6         int maxprofit = 0;
7         while (i < prices.length - 1) {
8             while (i < prices.length - 1 && prices[i] >=
prices[i + 1])
9                 i++;
10            valley = prices[i];
11            while (i < prices.length - 1 && prices[i] <=
prices[i + 1])
12                i++;
13            peak = prices[i];
14            maxprofit += peak - valley;
15        }
16        return maxprofit;
17    }
18 }
```

This solution follows the logic used in Approach 2 itself, but with only a slight variation. In this case, instead of looking for every peak following a valley, we can simply go on crawling over the slope and keep on adding the profit obtained from every consecutive transaction. In the end, we will be using the peaks and valleys effectively, but we need not track the costs corresponding to the peaks and valleys along with the maximum profit, but we can directly keep on adding the difference between the consecutive numbers of the array if the second number is larger than the first one, and at the total sum we obtain will be the maximum profit. This approach will simplify the solution. This can be made clearer by taking this example: [1, 7, 2, 3, 6, 7, 6, 7]

The graph corresponding to this array is:

From the above graph, we can observe that the sum A+B+CA+B+CA+B+C is equal to the difference D corresponding to the difference between the heights of the consecutive peak and valley.

```

1 class Solution {
2     public int maxProfit(int[] prices) {
3         int maxprofit = 0;
4         for (int i = 1; i < prices.length; i++) {
5             if (prices[i] > prices[i - 1])
6                 maxprofit += prices[i] - prices[i - 1];
7         }
8         return maxprofit;
9     }
10 }
```

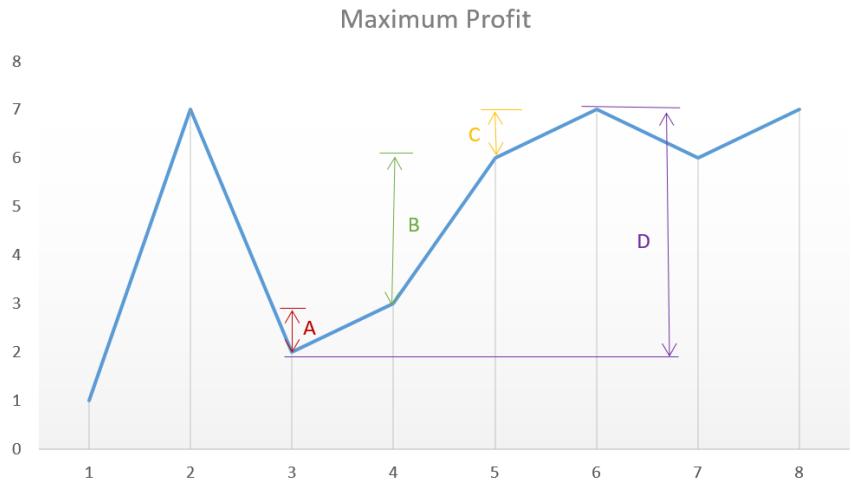


Figure 25.4: profit graph

25.3 Heap and Priority Queue

25.5 621. Task Scheduler (medium). Given a char array representing tasks CPU need to do. It contains capital letters A to Z where different letters represent different tasks. Tasks could be done without original order. Each task could be done in one interval. For each interval, CPU could finish one task or just be idle.

However, there is a non-negative cooling interval n that means between two **same tasks**, there must be at least n intervals that CPU are doing different tasks or just be idle. You need to return the **least** number of intervals the CPU will take to finish all the given tasks.

Example :

Input: tasks = ["A", "A", "A", "B", "B", "B"], n = 2

Output: 8

Explanation: A → B → idle → A → B → idle → A → B.

Analysis: we can approach the problem by thinking when we can get the least idle times? Whenever we put the same task together, they incurs largest idle time. Therefore, rule number 1: put different task next to each other whenever it is possible. However, consider the case: "A":6, "B":1, "C":1, "D":1, "E":1, if we simply do a round of using all of the available tasks in the decreasing order of their frequency, we get 'A, B, C, D, E, A, ?, A, ?, A, ?', here we end up with four '?', which represents idle. However, this is not the best solution. A better way that this is to use up the most frequent task as soon as its cooling time is finished. The new order is 'A, B, C, A, D, E, A, ?, A,

602 25. LINKED LIST, STACK, QUEUE, AND HEAP QUESTIONS (12%)

?, A, ?, A'. We end up with one less idle session. We can implement it with heapq due to the fact that it is more efficient compared with PriorityQueue().

Solution 1: heapq and idle cycle. We can use a map to get the frequency of each task, then we put their frequencies into a heapq, by using heapify function. When the list is not empty yet, for each idle cycle: which is $n+1$, we pop out items out and decrease its frequency and add time. (Actually, using PriorityQueue() here we will receive LTE.) We need $O(n)$ to iterate through the tasks list to get its frequency. Then heapify takes $O(26)$, each time, heappush takes $O(\log 26)$. This still makes the time complexity $O(n)$.

```

1 from collections import Counter
2 from queue import PriorityQueue
3 import heapq
4 def leastInterval(self, tasks, n):
5     c = Counter(tasks)
6     h = [-count for _, count in c.items()]
7     heapq.heapify(h)
8
9     ans = 0
10
11    while h:
12        temp = []
13        i = 0
14        while i <= n: # a cycle is n+1
15
16            if h:
17                c = heapq.heappop(h)
18                if c < -1:
19                    temp.append(c+1)
20                ans += 1
21                # if the queue is empty, we reached the end,
22                # need to break, no idle
23                if not h and not temp:
24                    break
25                i += 1
26            for c in temp:
27                heapq.heappush(h, c)
28
29    return ans

```

Solution 2: Use Sorting. Obversing Fig. 25.5, the actually time = idle time + total number of tasks. So, all we need to do is getting the idle time. And we start with the initial idle time which is (biggest frequency - 1)*(n). Then we travese the sorted list from the second item, and decrease the initial idle time. This gives us $O(n)$ time too. But the concept and coding is easier.

```

1 from collections import Counter
2 def leastInterval(self, tasks, n):
3     c = Counter(tasks)

```

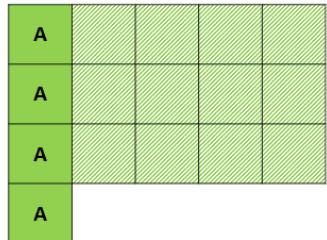


Figure 1

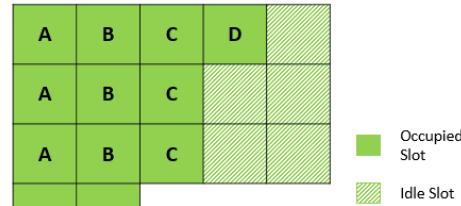


Figure 2

Figure 25.5: Task Scheduler, Left is the first step, the right is the one we end up with.

```

4     f = [count for _, count in c.items()]
5     f.sort(reverse=True)
6     idle_time = (f[0] - 1) * n
7
8     for i in range(1, len(f)):
9         c = f[i]
10        idle_time -= min(c, f[0]-1)
11    return idle_time + len(tasks) if idle_time > 0 else len(tasks)

```

604 25. *LINKED LIST, STACK, QUEUE, AND HEAP QUESTIONS (12%)*

26

String Questions (15%)

For the string problems, it can be divided into two categories: **one string** and **two strings pattern matching**.

For the one string problem, the first type is to do operations that meet certain requirements on a single string. (1). For the ad hoc easy string processing problems, we only need to read the requirement carefully and use basic programming skills, data structures, and sometimes requires us to be familiar with some string libraries like Re other than the basic built-in string functions. We list some LeetCode Problems of this type in Section 26.1. (2) There are also more challenging problems: including find the longest/shortest/ count substring and subsequence that satisfy certain requirements. Usually the subsequence is more difficult than the substring. In this chapter we would list the following types in Section 26.3

- Palindrome: A sequence of characters read the same forward and backward.
- Anagram: A word or phrase formed by rearranging the letters of a different word or phrase.
- Parentheses and others.

Application for Pattern Matching for two strings: Given two strings or two arrays, one is S, and the pattern P, The problems can be generalized to find pattern P in a string S, you would be given two strings. (1) If we do not care the order of the letters (anagram) in the pattern, then it is the best to use Sliding Window; This is detailed in Section 26.5 (2) If we care the order matters (identical to pattern), we use KMP. The problems of this type is listed in Section 26.6.

26.1 Ad Hoc Single String Problems

1. 125. Valid Palindrome
2. 65. Valid Number
3. 20. Valid Parentheses (use a stack to save left parenthe)
4. 214. Shortest Palindrome (KMP lookup table)
5. 5. Longest Palindromic Substring
6. 214 Shortest Palindrome , KMP lookup table, for example s=abba, constructed S = abba#abba),
7. 58. Length of Last Word(easy)

26.2 String Expression

1. 8. String to Integer (atoi) (medium)

26.3 Advanced Single String

For hard problem, reconstruct the problem to another so that it can be resolved by an algorithm that you know.

26.3.1 Palindrome

Palindrome is a sequence of characters read the same forward and backward. To identify if a sequence is a palindrome say "abba" we just need to check if $s == s[::-1]$. In the structure, if we know "bb" is palindrome, then "abba" should be palindrome if $s[0] == s[3]$. Due to this structure, in the problems with finding palindromic substrings, we can apply dynamic programming and other algorithms to fight back the naive solution.

To validate a palindrome we can use two pointers, one at the start, and the other at the end. We iterate them into the middle location.

1. 409. Longest Palindrome (*)
2. 9. Palindrome Number (*)
3. Palindrome Linked List (234, *)
4. Valid Palindrome (125, *)
5. Valid Palindrome II (680, *)
6. Largest Palindrome Product (479, *)

7. 647. Palindromic Substrings (medium, check)
8. Longest Palindromic Substring (5, **, check)
9. Longest Palindromic Subsequence(516, **)
10. Shortest Palindrome (214, ***)
11. Find the Closest Palindrome(564, ***)
12. Count Different Palindromic Subsequences(730, ***)
13. Palindrome Partitioning (131, **)
14. Palindrome Partitioning II (132, ***)
15. 266. Palindrome Permutation (Easy)
16. Palindrome Permutation II (267, **)
17. Prime Palindrome (866, **)
18. Super Palindromes (906, ***)
19. Palindrome Pairs (336, ***)
- 20.

26.1 **Valid Palindrome II (L680, *)**. Given a non-empty string s , you may delete **at most** one character. Judge whether you can make it a palindrome.

Example 1:

```
Input: "aba"
Output: True
```

Example 2:

```
Input: "abca"
Output: True
Explanation: You could delete the character 'c'.
```

Solution: Two Pointers. If we allow zero deletion, then it is a normal two pointers algorithm to check if the start i and the end j position has the same char. If we allow another time deletion is when the start and end char is not equal, we check if deleting $s[i]$ or $s[j]$, left $s(i+1, j)$ or $s(i, j-1)$ if they are palindrome.

```

1 def validPalindrome(self, s):
2     if not s:
3         return True
4
5     i, j = 0, len(s)-1
6     while i <= j:
7         if s[i] == s[j]:
8             i += 1
9             j -= 1
10        else:
11            left = s[i+1:j+1]
12            right = s[i:j]
13            return left == left[::-1] or right == right
14    return True

```

26.2 Palindromic Substrings(L647, **). Given a string, your task is to count how many palindromic substrings in this string. The substrings with different start indexes or end indexes are counted as different substrings even they consist of same characters.

Example 1:

```

Input: "abc"
Output: 3
Explanation: Three palindromic strings: "a", "b", "c".

```

Example 2:

```

Input: "aaa"
Output: 6
Explanation: Six palindromic strings: "a", "a", "a", "aa",
              "aa", "aaa".

```

Solution 1: Dynamic Programming. First, we use $dp[i][j]$ to denotes if the substring $s[i:j]$ is a palindrome or not. Thus, we have a matrix of size $n \times n$. We can apply a simple example “aaa”.

	` ` aaa "		
	0	1	2
0	1	1	1
1	0	1	1
2	0	0	1

From the example, first, we know this matrix would only have valid value at the upper part due to $i \leq j$. Because if $j-i \geq 3$ which means the length is larger or equals to 3, $dp[i][j] = 1$ if $s[i] == s[j]$ and $dp[i+1][j-1] == 1$. Compare $i:i+1, j:j-1$. This means we need to iterate i reversely and j incrementally.

```

1 def countSubstrings(self, s):
2     """

```

```

3     :type s: str
4     :rtype: int
5     """
6     n = len(s)
7     dp = [[0 for _ in range(n)] for _ in range(n)] # if
from i to j is a palindrome
8     res = 0
9     for i in range(n-1, -1, -1):
10        for j in range(i, n):
11            if j-i > 2: #length >=3
12                dp[i][j] = (s[i]==s[j] and dp[i+1][j-1])
13            else:
14                dp[i][j] = (s[i]==s[j]) #length 1 and 2
15            if dp[i][j]:
16                res += 1
17     return res

```

Range Type Dynamic Programming. A slightly different way to fill out the matrix is:

```

1 def countSubstrings(self, s):
2     if not s:
3         return 0
4
5     rows = len(s)
6     dp = [[0 for col in range(rows)] for row in range(rows)]
7     ans = 0
8     for i in range(0, rows):
9         dp[i][i] = 1
10        ans += 1
11
12    for l in range(2, rows+1): #length of substring
13        for i in range(0, rows-l+1): #start 0, end len-l+1
14            j = i+l-1
15            if j > rows:
16                continue
17            if s[i] == s[j]:
18                if j-i > 2:
19                    dp[i][j] = dp[i+1][j-1]
20                else:
21                    dp[i][j] = 1
22            ans += dp[i][j]
23
24    return ans

```

Solution 2: Center Expansion. For $s[0]='a'$, it is center at 0, $s[0:2]='aa'$, is center between 0 and 1, $s[1]='a'$, $s[0:3]='aaa'$, center at 1. $s[1:3]='aa'$ is center between 1 and 2, for $s[3]='a'$, is center at 2. There for our centers goes from: The time complexity if $O(n^2)$.

```

left = 0, right = 0, i = 0, i/2 = 0, i%2 = 0
left = 0, right = 1, i = 1, i/2 = 0, i%2 = 1
left = 1, right = 1, i = 2, i/2 = 1, i%2 = 0

```

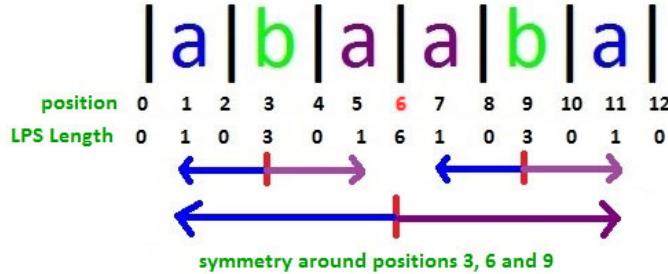


Figure 26.1: LPS length at each position for palindrome.

```
left = 1, right = 2, i = 3, i/2 = 1, i%2 = 1
left = 2, right = 2, i = 4, i/2 = 2, i%2 = 0
```

```
1 def countSubstrings(self, S):
2     n = len(S)
3     ans = 0
4     for i in range(2*n-1):
5         l = int(i/2)
6         r = l + i%2
7         while l >= 0 and r < n and S[l] == S[r]:
8             ans += 1
9             l -= 1
10            r += 1
11    return ans
```

Solution 3: Manacher's Algorithm. In the center expansion, we can save the result according to the position i . We can see from position 6, the LPS table is symmetric, what Manacher's Algorithm do is to identify around the center of a palindrome, when it will be symmetric and when it wont (in case at position 3, for immediate left and right (2, 4) is symmetric, but not (0, 5)). This is distinguished by the LPS length at position 3. only $(i-d, i, i+d)$ will be symmetric. The code for Python 2 is given: and try to understand later???

```
1 def manachers(S):
2     A = '@#' + '#' .join(S) + '#$'
3     Z = [0] * len(A)
4     center = right = 0
5     for i in xrange(1, len(A) - 1):
6         if i < right:
7             Z[i] = min(right - i, Z[2 * center - i])
8             while A[i + Z[i] + 1] == A[i - Z[i] - 1]:
9                 Z[i] += 1
10                if i + Z[i] > right:
11                    center, right = i, i + Z[i]
12    return Z
13
14 return sum((v+1)/2 for v in manachers(S))
```

26.3 Longest Palindromic Subsequence (L516, **). Given a string s , find the longest palindromic subsequence's length in s . You may assume that the maximum length of s is 1000.

```
Example 1:  
Input :  
"bbbab"  
Output :  
4  
One possible longest palindromic subsequence is "bbbb".  
  
Example 2:  
Input :  
"cbbd"  
Output :  
2  
One possible longest palindromic subsequence is "bb".
```

Solution: Range Type Dynamic Programming. We use $dp[i][j]$ to denote the maximum palindromic subsequence of $s(i,j)$. Like the substring palindrome, we only need to fill out the upper bound of the matrix. Let us dismentle the problems into different length of substring:

```
L=2: bb bb ba ab i=0..n-L+1, j=i+L-1  
L=3: bbb bba bab, if s[i] == s[j], Yes: dp[i][j] = dp[i+1][j-1]+2, which we obtained from last L2, No: dp[i][j] = max(dp[i+1][j], dp[i][j-1])  
L=4, bbba, bbab  
L=5, bbbb
```

The process will be controled by the range of the length of substring. And we fill out the matrix in the following way: this is a ranging type of dynamic programming.

```
[[1, 2, 0, 0, 0], [0, 1, 2, 0, 0], [0, 0, 1, 1, 0], [0, 0, 0, 1, 1], [0, 0, 0, 0, 1]]  
[[1, 2, 3, 0, 0], [0, 1, 2, 2, 0], [0, 0, 1, 1, 3], [0, 0, 0, 1, 2], [0, 0, 0, 0, 1]]  
[[1, 2, 3, 3, 0], [0, 1, 2, 2, 3], [0, 0, 1, 1, 3], [0, 0, 0, 1, 2], [0, 0, 0, 0, 1]]  
[[1, 2, 3, 3, 4], [0, 1, 2, 2, 3], [0, 0, 1, 1, 3], [0, 0, 0, 1, 2], [0, 0, 0, 0, 1]]
```

```
1 def longestPalindromeSubseq(self, s):  
2     if not s:  
3         return 0  
4     if s == s[::-1]:  
5         return len(s)  
6  
7     rows = len(s)  
8     dp = [[0 for col in range(rows)] for row in range(rows)]
```

```

9   for i in range(0,rows):
10    dp[i][i] = 1
11
12   for l in range(2, rows+1): #use a space
13     for i in range(0,rows-l+1): #start 0, end len -l+1
14       j = i+l-1
15       if j > rows:
16         continue
17       if s[i] == s[j]:
18         dp[i][j] = dp[i+1][j-1]+2
19       else:
20         dp[i][j] = max(dp[i][j-1], dp[i+1][j])
21   return dp[0][rows-1]

```

26.3.2 Calculator

In this section, for basic calculator, we have operators '+', '−', '*', '/', and parentheses. Because of ('+', '−') and ('*', '/') has different priority, and the parentheses change the priority too. The basic step is to obtain the integers digit by digit from the string. And, if the previous sign is '−', we make sure we get a negative number. Given a string expression: $(a+b/c)*(d-e)+((f-g)-(h-i))+(j-k)$. The rule here is to deduct this to:

$$\frac{(a + \frac{b}{c})_a * (d - e)_d}{a} + \frac{((f - g)_f - (h - i)_h)}{f} + \frac{(j - k)_j}{j} \quad (26.1)$$

The rules are: 1) Reduce the '*' and '/': And we handle it when we encounter the following operator or at the end of the string. Because, when we encounter a sign(operator), we check the previous sign, if the previous sign is '/' or '*', we compute the previous number with current number to reduce it into one. 2) Reduce the parentheses into one: $(d-e)$ is reduced to d , and because the previous sign is '*', it is further combined with a and become a . Thus, if we save the reduced result into a stack, there will be $[a, f, j]$, we just need to sum over. thus to avoid the boundary condition, we can add '+' at the end. In the later part, we will explain more about how to deal with the above two kinds of reduce. There are different levels of calculators:

1. '+', '−', w/o parentheses: e.g., $a+b+c$, or $a-b-c$.

```

presign = '+', num=0, for the digits, stack for saving
either negative or positive integer
1. iterate through the char:
   if a digit: obtain the integer
   else if c in ['+', '-'] or c is the last char:
     if presign == '-':
       num = -num
     stack.append(num)
     num = 0
     presign = c

```

```
2. sum over the positive and negative values in the
stack
```

2. '+', '-', with parentheses: e.g. a-b-c vs a-(b-c-d). To handle the parentheses, we need to think of (b-c-d) as a single integer. When we encounter the left parenthesis we save its state: the previous sign and '('; when encountering ')', we do a sum over in the stack till we pop out the previous ')'. And we recover its state: the previous sign and the num.

```
if c == '(':
    stack.append(presign)
    stack.append('(')
    presign = '+'
    num = 0
else if c in ['+', '-', ')']: # if its operator or ')'
    if presign == '-':
        num = -num
    if c == ')':
        sum over in the stack till top is '(',
        restore the state
    else:
        stack.append(num)
        num=0, presign = c
```

3. '+', '-', '**', '/', w/o parentheses: This is similar to Case 1, other than the '**', '-'. For example, a-b/c/d*e. When we are at c, we compute the pop the top element in the stack and compute (-b/c)=f, and append f into the stack. When we are at d, similarly, we compute (f/d)=g, and append g into the stack.

```
1. iterate through the char:
    if a digit: obtain the integer
    if c in ['+', '-', '*', '/'] or c is the last char:
        if presign == '-':
            num = -num
        # we reduce the current num with previous
        elif presign in ['*', '/']:
            num = operator(stack.pop(), presign, num)
            stack.append(num)
            num = 0
            presign = c
2. sum over the positive and negative values in the
stack
```

4. '+', '-', '**', '/', with parentheses. It is a combination of the previous cases, so I am not giving code here.

26.4 Basic Calculator (L224, *).** Implement a basic calculator to evaluate a simple expression string. The expression string may contain open (and closing parentheses), the plus + or minus sign -, non-negative integers and empty spaces.

Example 1:

Input: "1 + 1"

Output: 2

Example 2:

Input: " 2-1 + 2 "

Output: 3

Example 3:

Input: "(1+(4+5+2)-3)+(6+8)"

Output: 23

Stack for Parentheses. Suppose firstly we don't consider the parentheses, then it is linear iterating each char and handle the digits and the sign. The code are the first if and elif in the following Python code. Now, to think of the parentheses, it does affect the result: 2-(5-6). With and without parentheses give 3 and -9 for answer. When we encounter a '(', we need to reset ans and the sign, plus we need to save the previous ans and sign, at here it is (2, -). Then when we encounter a ')', we first collect the answer from last '(' to current ')'. And, we need to sum up the answer before ')'.

```
(1+(4+5+2)-3)+(6+8)
at (: stack = [0, +]
at second '(': stack = [0, +, 1, +]
at first ')': ans=11, pop out [1, +], ans = 12, +
at second ')': ans = 9, pop out [0, +], ans = 9
at third '(': ans = 9, +, stack = [9,+], reset ans = 0,
sign = '+'
```

```
1 def calculate(self, s):
2     s = s + '+'
3     ans = num = 0 #num is to get each number
4     sign = '+'
5     stack = collections.deque()
6     for c in s:
7         if c.isdigit(): #get number
8             num = 10*num + int(c)
9         elif c in ['-','+', ')']:
10             if sign == '-':
11                 num = -num
12             if c == ')':
13                 while stack and stack[-1] != '(':
14                     num += stack.pop()
15                 stack.pop()
16                 sign = stack.pop()
17             else:
```

```
18             stack.append(num)
19             num = 0
20             sign = c
21     elif c == '(': # left parathese, put the current
22         ans and sign in the stack
23             stack.append(sign)
24             stack.append('(')
25             num = 0
26             sign = '+'
27
28     while stack:
29         ans += stack.pop()
30
31     return ans
```

26.5 **Basic Calculator III (L772, ***)**. Implement a basic calculator to evaluate a simple expression string. The expression string may contain open (and closing parentheses), the plus + or minus sign -, **non-negative** integers and empty spaces . The expression string contains only non-negative integers, +, -, *, / operators , open (and closing parentheses) and empty spaces . The integer division should truncate toward zero. You may assume that the given expression is always valid. All intermediate results will be in the range of [-2147483648, 2147483647].

Some examples:

```

"1 + 1" = 2
" 6-4 / 2 " = 4
"2*(5+5*2)/3+(6/2+8)" = 21
"(2+6* 3+5- (3*14/7+2)*5)+3"=-12

```

Solution: Case 4

```
1 def calculate(self, s):
2     ans = num = 0
3     stack = collections.deque()
4     n = len(s)
5     presign = '+'
6     s = s+''
7     def op(pre, op, cur):
8         if op == '*':
9             return pre*cur
10        if op == '/':
11            return -abs(pre)//cur if pre < 0 else pre//cur
12    for i, c in enumerate(s):
13        if c.isdigit():
14            num = 10*num + int(c)
15        elif c in ['+', '-', '*', '/', ')']:
16            if presign == '-':
17                num = -num
18            elif presign in ['*', '/']:
19                num = op(stack.pop(), presign, num)
20            presign = c
21        stack.append(num)
22    return ans
```

```

20         if c == ')': # reduce to one number, and
21             restore the state
22                 while stack and stack[-1] != '(':
23                     num += stack.pop()
24                     stack.pop() # pop out '('
25                     presign = stack.pop()
26             else:
27                 stack.append(num)
28                 num = 0
29                 presign = c
30             elif c == '(': # save state, and restart a new
31                 process
32                 stack.append(presign)
33                 stack.append(c)
34                 presign = '+'
35                 num = 0
36
37             ans = 0
38             while stack:
39                 ans += stack.pop()
40
41     return ans

```

26.6 227. Basic Calculator II (exercise)

26.3.3 Others

Possible methods: two pointers, one loop+two pointers

26.4 Exact Matching: Sliding Window and KMP

Exact Pattern Matching

14. Longest Common Prefix (easy)

26.5 Anagram Matching: Sliding Window

However, if the question is to find all anagrams of the pattern in string S.
For example: 438. Find All Anagrams in a String

Example 1:

```

1 Input:
2 s: "cbaebabacd" p: "abc"
3
4 Output:
5 [0, 6]

```

Explanation: The substring with start index = 0 is "cba", which is an anagram of "abc". The substring with start index = 6 is "bac", which is an anagram of "abc".

Python code with sliding window:

```

1 def findAnagrams(self, s, p):
2     """
3         :type s: str
4         :type p: str
5         :rtype: List[int]
6     """
7     if len(s) < len(p) or not s:
8         return []
9     #frequency table
10    table = {}
11    for c in p:
12        table[c] = table.get(c, 0) + 1
13
14    begin, end = 0, 0
15    r = []
16    counter = len(table)
17    while end < len(s):
18        end_char = s[end]
19        if end_char in table.keys():
20            table[end_char] -= 1
21            if table[end_char] == 0:
22                counter -= 1
23
24        #go to longer string, from A, AD,
25        end += 1
26
27        while counter == 0: #we have the same char in the
28            window, start to trim it
29            #save the best result, just to save the
30            begining
31            if end - begin == len(p):
32                r.append(begin)
33            #move the window forward
34            start_char = s[begin]
35            if start_char in table: #reverse the count
36                table[start_char] += 1
37                if table[start_char] == 1: #only increase when
38                    begin += 1
39                    counter += 1
40
41    return r

```

26.6 Exact Matching

26.6.1 Longest Common Subsequence

26.7 Exercise

26.7.1 Palindrome

26.1 EXERCISES

- 1. Valid Palindrome (L125, *).** Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.
Note: For the purpose of this problem, we define empty string as valid palindrome.

Example 1:

Input: "A man, a plan, a canal: Panama"
 Output: true

Example 2:

Input: "race a car"
 Output: false

- 2. Longest Palindrome (L409, *).** Given a string which consists of lowercase or uppercase letters, find the length of the longest palindromes that can be built with those letters. This is case sensitive, for example "Aa" is not considered a palindrome here.

Example:

Input:
 "abcccccdd"

Output:
 7

Explanation:

One longest palindrome that can be built is "dccaccd", whose length is 7.

- 3. Longest Palindromic Substring(L5, **).** Given a string s, find the longest palindromic substring in s. You may assume that the maximum length of s is 1000.

Example 1:

Input: "babad"
 Output: "bab"
 Note: "aba" is also a valid answer.

Example 2:

Input: "cbbd"
 Output: "bb"

27

Tree Questions(10%)

The purpose of the

27.1 Binary Search Tree

In computer science, a **search tree** is a tree data structure used for locating specific keys from within a set. In order for a tree to function as a search tree, the key for each node must be greater than any keys in subtrees on the left and less than any keys in subtrees on the right.

The advantage of search trees is their efficient search time ($O(\log n)$) given the tree is reasonably balanced, which is to say the leaves at either end are of comparable depths as we introduced the **balanced binary tree**.

The search tree data structure supports many dynamic-set operations, including **Search** for a key, minimum or maximum, predecessor or successor, **insert** and **delete**. Thus, a search tree can be both used as a dictionary and a priority queue.

A binary search tree (BST) is a search tree with children up to two. There are three possible ways to properly define a BST, and we use l and r to represent the left and right child of node x : 1) $l.key \leq x.key < r.key$, 2) $l.key < x.key \leq r.key$, 3) $l.key < x.key < r.key$. In the first and second definition, our resulting BST allows us to have duplicates, while not in the case of the third definition. One example of BST without duplicates is shown in Fig 27.1.

Operations

When looking for a key in a tree (or a place to insert a new key), we traverse the tree from root to leaf, making comparisons to keys stored in the

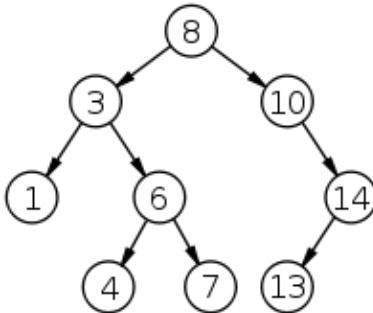


Figure 27.1: Example of Binary search tree of depth 3 and 8 nodes.

nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each SEARCH, INSERT or DELETE takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

In order to build a BST, we need to INSERT a series of elements in the tree organized by the searching tree property, and in order to INSERT, we need to SEARCH the position to INSERT this element. Thus, we introduce these operations in the order of SEARCH, INSERT and GENERATE.

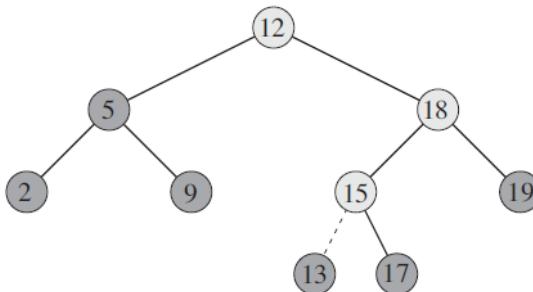


Figure 27.2: The lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

SEARCH There are two different implementations for SEARCH: recursive and iterative.

```

1 # recursive searching
2 def search(root ,key):
3     # Base Cases: root is null or key is present at root
4         if root is None or root.val == key:
  
```

```

5     return root
6
7 # Key is greater than root's key
8 if root.val < key:
9     return search(root.right, key)
10
11 # Key is smaller than root's key
12 return search(root.left, key)

```

Also, we can write it in an iterative way, which helps us save the heap space:

```

1 # iterative searching
2 def iterative_search(root, key):
3     while root is not None and root.val != key:
4         if root.val < key:
5             root = root.right
6         else:
7             root = root.left
8     return root

```

INSERT Assuming we are inserting a node 13 into the tree shown in Fig 27.2. A new key is always inserted at leaf (there are other ways to insert but here we only discuss this one way). We start searching a key from root till we hit an empty node. Then we new a TreeNode and insert this new node either as the left or the child node according to the searching property. Here we still shows both the recursive and iterative solutions.

```

1 # Recursive insertion
2 def insertion(root, key):
3     if root is None:
4         root = TreeNode(key)
5         return root
6     if root.val < key:
7         root.right = insertion(root.right, key)
8     else:
9         root.left = insertion(root.left, key)
10    return root

```

The above code needs return value and reassign the value for the right and left every time, we can use the following code which might looks more complex with the if condition but works faster and only assign element at the end.

```

1 # recursive insertion
2 def insertion(root, val):
3     if root is None:
4         root = TreeNode(val)
5         return
6     if val > root.val:
7         if root.right is None:
8             root.right = TreeNode(val)
9         else:
10            insertion(root.right, val)

```

```

11     else:
12         if root.left is None:
13             root.left = TreeNode(val)
14         else:
15             insertion(root.left, val)

```

We can search the node iteratively and save the previous node. The while loop would stop when hit at an empty node. There will be three cases in the case of the previous node.

1. The previous node is None, which means the tree is empty, so we assign a root node with the value
2. The previous node has a value larger than the key, means we need to put key as left child.
3. The previous node has a value smaller than the key, means we need to put key as right child.

```

1 # iterative insertion
2 def iterativeInsertion(root, key):
3     pre_node = None
4     node = root
5     while node is not None:
6         pre_node = node
7         if key < node.val:
8             node = node.left
9         else:
10            node = node.right
11    # we reached to the leaf node which is pre_node
12    if pre_node is None:
13        root = TreeNode(key)
14    elif pre_node.val > key:
15        pre_node.left = TreeNode(key)
16    else:
17        pre_node.right = TreeNode(key)
18    return root

```

BST Generation First, let us declare a node as BST which is the root node. Given a list, we just need to call INSERT for each element. The time complexity can be $O(n \log n)$.

```

1 datas = [8, 3, 10, 1, 6, 14, 4, 7, 13]
2 BST = None
3 for key in datas:
4     BST = iterativeInsertion(BST, key)
5 print(LevelOrder(BST))
6 # output
7 # [8, 3, 10, 1, 6, 14, 4, 7, 13]

```

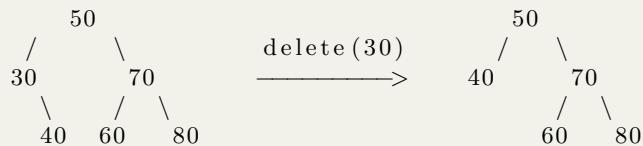
DELETE Before we start to check the implementation of DELETE, I would suggest the readers to read the next subsection—the Features of BST at first, and then come back here to finish this paragraph.

When we delete a node, three possibilities arise.

- 1) Node to be deleted is leaf: Simply remove from the tree.



- 2) Node to be deleted has only one child: Copy the child to the node and delete the child



- 3) Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

Features of BST

Minimum and Maximum The operation is similar to search, to find the minimum, we always traverse on the left subtree. For the maximum, we just need to replace the “left” with “right” in the key word. Here the time complexity is the same $O(lgn)$.

```

1 # recursive
2 def get_minimum(root):
3     if root is None:
4         return None
5     if root.left is None: # a leaf or node has no left subtree
6         return root
7     if root.left:
8         return get_minimum(root.left)
9
10 # iterative
11 def iterative_get_minimum(root):
12     while root.left is not None:
13         root = root.left
  
```

```
14     return root
```

Also, sometimes we need to search two additional items related to a given node: successor and predecessor. The structure of a binary search tree allows us to determine the successor or the predecessor of a tree without ever comparing keys.

Successor of a Node A successor of node x is the smallest item in the BST that is strictly greater than x . It is also called in-order successor, which is the next node in Inorder traversal of the Binary Tree. Inoreder Successor is None for the last node in inorder traversal. If our TreeNode data structure has a parent node.

Use parent node: the algorihtm has two cases on the basis of the right subtree of the input node.

For the right subtree of the node:

- 1) If it is not None, then the successor is the minimum node in the right subtree. e.g. for node 12, $\text{successor}(12) = 13 = \min(12.\text{right})$
- 2) If it is None, then the successor is one of its ancestors. We traverse up using the parent node until we find a node which is the left child of its parent. Then the parent node here is the successor. e.g. $\text{successor}(2)=5$

The Python code is provided:

```
1 def Successor(root , n):
2 # Step 1 of the above algorithm
3     if n.right is not None:
4         return get_minimum(n.right)
5 # Step 2 of the above algorithm
6 p = n.parent
7 while p is not None:
8     if n == p.left :# if current node is the left child node,
9         then we found the successor , p
10        return p
11    n = p
12    p = p.parent
13 return p
```

However, if it happens that your tree node has no parent defined, which means you can not traverse back its parents. We only have one option. Use the inorder tree traversal, and find the element right after the node.

For the right subtree of the node:

- 1) If it is not None, then the successor is the minimum node in the right subtree. e.g. for node 12, $\text{successor}(12) = 13 = \min(12.\text{right})$
- 2) If it is None, then the successor is one of its ancestors. We traverse down from the root till we find current node, the node in advance of current node is the successor. e.g. $\text{successor}(2)=5$

```

1 def SuccessorInorder(root , n):
2     # Step 1 of the above algorithm
3     if n.right is not None:
4         return get_minimum(n.right)
5     # Step 2 of the above algorithm
6     succ = None
7     while root is not None:
8
9         if n.val > root.val:
10            root = root.right
11        elif n.val < root.val:
12            succ = root
13            root = root.left
14        else: # we found the node, no need to traverse
15            break
16    return succ

```

Predecessor of A Node A predecessor of node x on the other side, is the largest item in BST that is strictly smaller than x . It is also called in-order predecessor, which denotes the previous node in Inorder traversal of BST. e.g. for node 14, predecessor(14)=12= max(14.left). The same searching rule applies, if node x 's left subtree exists, we return the maximum value of the left subtree. Otherwise we traverse back its parents, and make sure it is the right subtree, then we return the value of its parent, otherwise the reversal traverse keeps going.

```

1 def Predecessor (root , n):
2 # Step 1 of the above algorithm
3     if n.left is not None:
4         return get_maximum(n.left)
5 # Step 2 of the above algorithm
6 p = n.parent
7 while p is not None:
8     if n == p.right :# if current node is the right node, parent
9         is smaller
10        return p
11     n = p
12     p = p.parent
13 return p

```

The worst case to find the successor or the predecessor of a BST is to search the height of the tree: include the one of the subtrees of the current node, and go back to all the parents and greatparents of this code, which makes it the height of the tree. The expected time complexity is $O(lgn)$. And the worst is when the tree line up and has no branch, which makes it $O(n)$.

Now we put a table here to summarize the space and time complexity for each operation.

Table 27.1: Time complexity of operations for BST in big O notation

Algorithm	Average	Worst Case
Space	$O(n)$	$O(n)$
Search	$O(lgn)$	$O(n)$
Insert	$O(lgn)$	$O(n)$
Delete	$O(lgn)$	$O(n)$

27.2 Segment Tree

Segment Tree is a static full binary tree similar to heap that is used for storing the intervals or segments. ‘Static’ here means once the data structure is build, it can not be modified or extended. Segment tree is a data structure that can efficiently answer numerous *dynamic range queries* problems (in logarithmic time) like finding minimum, maximum, sum, greatest common divisor, least common denominator in array. The “dynamic” means there are constantly modifications of the value of elements (not the tree structure). For instance, given a problem to find the index of the minimum/maximum/-sum of all elements in an given range of an array: [i:j].

Definition Consider an array A of size n and a corresponding Segment Tree T (here a range [0, n-1] in A is represented as A[0:N-1]):

1. The root of T represents the whole array A[0:N-1].
2. Each internal node in the Segment Tree T represents the interval of A[i:j] where $0 < i < j < n$.
3. Each leaf in T represents a single element A[i], where $0 \leq i < N$.
4. If the parent node is in range [i, j], then we separate this range at the middle position $m = (i + j)/2$, the left child takes range [i, m], and the right child take the interval of [m+1, j].

Because in each step of building the segment tree, the interval is divided into two halves, so the height of the segment tree will be $\log N$. And there will be totally N leaves and N-1 number of internal nodes, which makes the total number of nodes in segment tree to be $2N - 1$ and make the segment tree a *full binary tree*.

Here, we use the Range Sum Query (RSQ) problem to demonstrate how segment tree works:

27.1 307. Range Sum Query - Mutable (medium). Given an integer array nums, find the sum of the elements between indices i and j ($i \leq j$), inclusive. The *update(i, val)* function modifies nums by updating the element at index i to val.

Example :

```
Given nums = [1, 3, 5]
sumRange(0, 2) -> 9
update(1, 2)
sumRange(0, 2) -> 8
```

Note:

1. The array is only modifiable by the update function.
2. You may assume the number of calls to update and sumRange function is distributed evenly.

Solution: Brute-Force. There are several ways to solve the RSQ. The **brute-force solution** is to simply iterate the array from index i to j to sum up the elements and return its corresponding index. And it gives $O(n)$ per query, such algorithm maybe infeasible if queries are constantly required. Because the update and query action distributed evenly, it still gives $O(n)$ time complexity and $O(n)$ in space, which will get LET error.

Solution: Segment Tree. With Segment Tree, we can store the TreeNode's val as the sum of elements in its corresponding interval. We can define a TreeNode as follows:

```
1 class TreeNode:
2     def __init__(self, val, start, end):
3         self.val = val
4         self.start = start
5         self.end = end
6         self.left = None
7         self.right = None
```

As we see in the process, it is actually not necessary if we save the size of the array, we can decide the start and end index of each node on-the-fly and saves space.

Build Segment Tree. Because the leaves of the tree is a single element, we can use divide and conquer to build the tree recursively. For a given node, we first build and return its left and right child(including calculating its sum) in advance in the ‘divide’ step, and in the ‘conquer’ step, we calculate this node’s sum using its left and right child’s sum, and set its left and right child. Because there are totally $2n - 1$ nodes, which makes the time and space complexity $O(n)$.

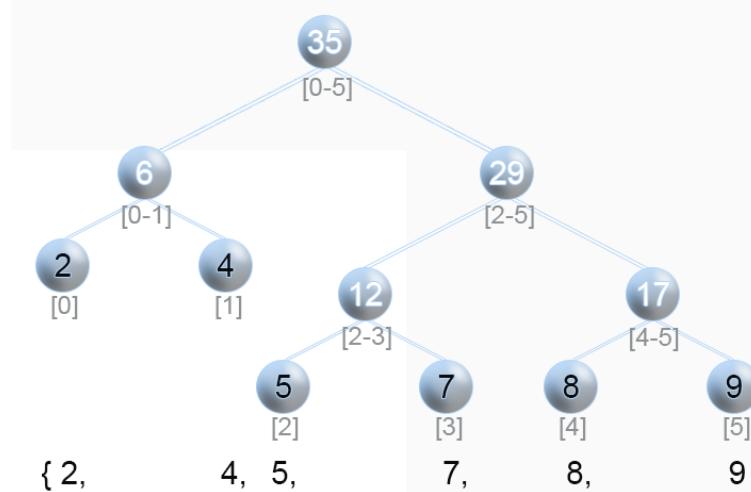
Segmented tree for array $\{2, 4, 5, 7, 8, 9\}$

Figure 27.3: Illustration of Segment Tree.

```

1 def __buildSegmentTree(self, nums, s, e): #start index and
2     end index
3     if s > e:
4         return None
5     if s == e:
6         return self.TreeNode(nums[s])
7
8     m = (s + e)//2
9     # divide
10    left = self.__buildSegmentTree(nums, s, m)
11    right = self.__buildSegmentTree(nums, m+1, e)
12
13    # conquer
14    node = self.TreeNode(left.val + right.val)
15    node.left = left
16    node.right = right
17    return node

```

Update Segment Tree. Updating the value at index i is like searching the tree for leaf node with range $[i, i]$. We just need to recalculate the value of the node in the path of the searching. This operation takes $O(\log n)$ time complexity.

```

1 def __updateNode(self, i, val, root, s, e):
2     if s == e:
3         root.val = val
4         return
5     m = (s + e)//2
6     if i <= m:

```

```

7     self._updateNode(i, val, root.left, s, m)
8 else:
9     self._updateNode(i, val, root.right, m+1, e)
10    root.val = root.left.val + root.right.val
11    return

```

Range Sum Query. Each query range $[i, j]$, will be a combination of ranges of one or multiple ranges. For instance, as in the segment tree shown in Fig 27.3, for range $[2, 4]$, it will be combination of $[2, 3]$ and $[4, 4]$. The process is similar to the updating, we starts from the root, and get its middle index m : 1) if $[i, j]$ is the same as $[s, e]$ that $i == s$ and $j == e$, then return the value, 2) if the interval $[i, j]$ is within range $[s, m]$ that $j <= m$, then we just search it in the left branch. 3) if $[i, j]$ in within range $[m+1, e]$ that $i > m$, then we search for the right branch. 4) else, we search both branch and the left branch has target $[i, m]$, and the right side has target $[m+1, j]$, the return value should be the sum of both sides. The time complexity is still $O(\log n)$.

```

1 def __rangeQuery(self, root, i, j, s, e):
2     if s > e or i > j:
3         return 0
4     if s == i and j == e:
5         return root.val if root is not None else 0
6
7     m = (s + e)//2
8
9     if j <= m:
10        return self.__rangeQuery(root.left, i, j, s, m)
11    elif i > m:
12        return self.__rangeQuery(root.right, i, j, m+1, e)
13    else:
14        return self.__rangeQuery(root.left, i, m, s, m) +
self.__rangeQuery(root.right, m+1, j, m+1, e)

```

The complete code is given:

```

1 class NumArray:
2     class TreeNode:
3         def __init__(self, val):
4             self.val = val
5             self.left = None
6             self.right = None
7
8         def __init__(self, nums):
9             self.n = 0
10            self.st = None
11            if nums:
12                self.n = len(nums)
13                self.st = self._buildSegmentTree(nums, 0, self.
n-1)
14

```

```

15     def update(self, i, val):
16         self._updateNode(i, val, self.st, 0, self.n -1)
17
18     def sumRange(self, i, j):
19         return self._rangeQuery(self.st, i, j, 0, self.n-1)

```

Segment tree can be used here to lower the complexity of each query to $O(\log n)$.

27.3 Trie for String

Definition Trie comes from the word reTrieval. In computer science, a trie, also called digital tree, radix tree or prefix tree which like BST is also a kind of search tree for finding substring in a text. We can solve string matching in $O(|T|)$ time, where $|T|$ is the size of our text. This purely algorithmic approach has been studied extensively in the algorithms: Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp. However, we entertain the possibility that multiple queries will be made to the same text. This motivates the development of data structures that preprocess the text to allow for more efficient queries. Such efficient data structure is Trie, which can do each query in $O(P)$, where P is the length of the pattern string. Trie is an ordered tree structure, which is used mostly for storing strings (like words in dictionary) in a compact way.

1. In a Trie, each child branch is labeled with letters in the alphabet Σ . Actually, it is not necessary to store the letter as the key, because if we order the child branches of every node alphabetically from left to right, the position in the tree defines the key which it is associated to.
2. The root node in a Trie represents an empty string.

Now, we define a trie Node: first it would have a bool variable to denote if it is the end of the word and a children which is a list of 26 children TrieNodes.

```

1 class TrieNode:
2     # Trie node class
3     def __init__(self):
4         self.children = [None]*26
5         # isEndOfWord is True if node represent the end of the
6         # word
6         self.isEndOfWord = False

```

Compact Trie If we assign only one letter per edge, we are not taking full advantage of the trie's tree structure. It is more useful to consider compact or compressed tries, tries where we remove the one letter per edge constraint, and contract non-branching paths by concatenating the letters on

Compressed Trie

- Compress unary nodes, label edges by strings

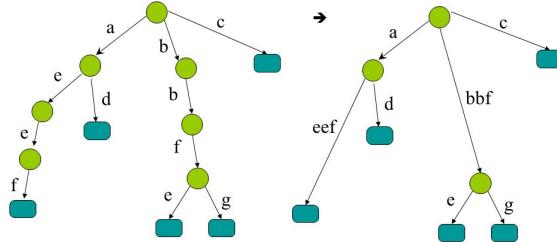


Figure 27.4: Trie VS Compact Trie

these paths. In this way, every node branches out, and every node traversed represents a choice between two different words. The compressed trie that corresponds to our example trie is also shown in Figure 27.4.

Operations: INSERT, SEARCH Both for INSERT and SEARCH, it takes $O(m)$, where m is the length of the word/string we want to insert or search in the trie. Here, we use an LeetCode problem as an example showing how to implement INSERT and SEARCH. Because constructing a trie is a series of INSERT operations which will take $O(n * m)$, n is the total numbers of words/strings, and m is the average length of each item. The space complexity for the non-compact Trie would be $O(N * |\Sigma|)$, where $|\Sigma|$ is the alphabetical size, and N is the total number of nodes in the trie structure. The upper bound of N is $n * m$.

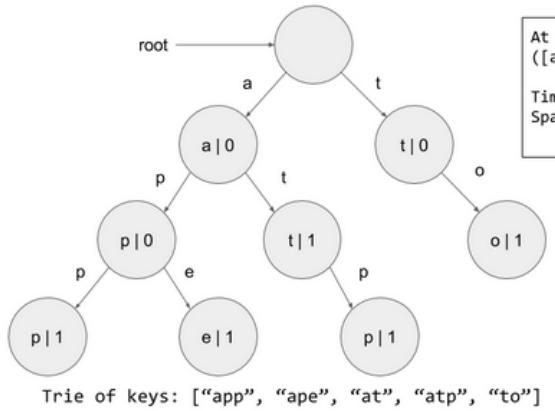


Figure 27.5: Trie Structure

27.1 208. Implement Trie (Prefix Tree) (medium). Implement a trie with insert, search, and startsWith methods.

```

1 Example:
2 Trie trie = new Trie();
3 trie.insert("apple");
4 trie.search("apple");    // returns true
5 trie.search("app");     // returns false
6 trie.startsWith("app"); // returns true
7 trie.insert("app");
8 trie.search("app");    // returns true

```

Note: You may assume that all inputs are consist of lowercase letters a-z. All inputs are guaranteed to be non-empty strings.

INSERT with INSERT operation, we woould be able to insert a given word in the trie, when traversing the trie from the root node which is a TrieNode, with each letter in world, if its corresponding node is None, we need to put a node, and continue. At the end, we need to set that node's endofWord variable to True. thereafter, we would have a new branch starts from that node constructed. For example, when we first insert "app" as shown in Fig 27.4, we would end up building branch "app", and with ape, we would add nodes "e" as demonstrated with red arrows.

```

1 def insert(self, word):
2     """
3     Inserts a word into the trie.
4     :type word: str
5     :rtype: void
6     """
7     node = self.root #start from the root node
8     for c in word:
9         loc = ord(c)-ord('a')
10        if node.children[loc] is None: # char does not
11            exist, new one
12            node.children[loc] = self.TrieNode()
13            # move to the next node
14            node = node.children[loc]
15        # set the flag to true
16        node.is_word = True

```

SEARCH For SEARCH, like INSERT, we traverse the trie using the letters as pointers to the next branch. There are three cases: 1) for word P, if it doesnt exist, but its prefix does exist, then we return False. 2) If we found a matching for all the letters of P, at the last node, we need to check if it is a leaf node where is_word is True. STARTWITH is just slightly different from SEARCH, it does not need to check that and return True after all letters matched.

```

1 def search(self, word):
2     node = self.root
3     for c in word:
4         loc = ord(c)-ord('a')
5         # case 1: not all letters matched
6         if node.children[loc] is None:
7             return False
8         node = node.children[loc]
9     # case 2
10    return True if node.is_word else False

11
1 def startWith(self, word):
2     node = self.root
3     for c in word:
4         loc = ord(c)-ord('a')
5         # case 1: not all letters matched
6         if node.children[loc] is None:
7             return False
8         node = node.children[loc]
9     # case 2
10    return True

```

Now complete the given Trie class with TrieNode and `__init__` function.

```

1 class Trie:
2     class TrieNode:
3         def __init__(self):
4             self.is_word = False
5             self.children = [None] * 26 #the order of the
node represents a char
6
7     def __init__(self):
8         """
9             Initialize your data structure here.
10            """
11        self.root = self.TrieNode() # root has value None

```

27.1 336. Palindrome Pairs (hard). Given a list of unique words, find all pairs of distinct indices (i, j) in the given list, so that the concatenation of the two words, i.e. $\text{words}[i] + \text{words}[j]$ is a palindrome.

```

1 Example 1:
2
3 Input: ["abcd", "dcba", "lls", "s", "sssll"]
4 Output: [[0,1],[1,0],[3,2],[2,4]]
5 Explanation: The palindromes are ["dcbaabed", "abcddcba", "
slls", "llssssll"]
6
7 Example 2:
8
9 Input: ["bat", "tab", "cat"]
10 Output: [[0,1],[1,0]]
11 Explanation: The palindromes are ["battab", "tabbat"]

```

Solution: One Forward Trie and Another Backward Trie. We start from the naive solution, which means for each element, we check if it is palindrome with all the other strings. And from the example 1, [3,3] can be a pair, but it is not one of the outputs, which means this is a combination problem, the time complexity is $C_n C_{n-1}$, and multiply it with the average length of all the strings, we make it m , which makes the complexity to be $O(mn^2)$. However, we can use Trie Structure,

```

1  from collections import defaultdict
2
3
4  class Trie:
5      def __init__(self):
6          self.links = defaultdict(self.__class__)
7          self.index = None
8          # holds indices which contain this prefix and whose
9          # remainder is a palindrome
10         self.pali_indices = set()
11
12     def insert(self, word, i):
13         trie = self
14         for j, ch in enumerate(word):
15             trie = trie.links[ch]
16             if word[j+1:] and is_palindrome(word[j+1:]):
17                 trie.pali_indices.add(i)
18         trie.index = i
19
20     def is_palindrome(word):
21         i, j = 0, len(word) - 1
22         while i <= j:
23             if word[i] != word[j]:
24                 return False
25             i += 1
26             j -= 1
27         return True
28
29
30     class Solution:
31         def palindromePairs(self, words):
32             '''Find pairs of palindromes in O(n*k^2) time and O
33             (n*k) space.'''
34             root = Trie()
35             res = []
36             for i, word in enumerate(words):
37                 if not word:
38                     continue
39                 root.insert(word[::-1], i)
40             for i, word in enumerate(words):
41                 if not word:
42                     continue
43                 trie = root

```

```

43     for j, ch in enumerate(word):
44         if ch not in trie.links:
45             break
46         trie = trie.links[ch]
47         if is_palindrome(word[j+1:]) and trie.index
48             is not None and trie.index != i:
49                 # if this word completes to a
50                 # palindrome and the prefix is a word, complete it
51                 res.append([i, trie.index])
52             else:
53                 # this word is a reverse suffix of other
54                 # words, combine with those that complete to a palindrome
55                 for pali_index in trie.pali_indices:
56                     if i != pali_index:
57                         res.append([i, pali_index])
58             if '' in words:
59                 j = words.index('')
60                 for i, word in enumerate(words):
61                     if i != j and is_palindrome(word):
62                         res.append([i, j])
63                         res.append([j, i])
64
65     return res

```

Solution2: Moreover, there are always more clever ways to solve these problems. Let us look at a clever way: abcd, the prefix is ". 'a', 'ab', 'abc', 'abcd', if the prefix is a palindrome, so the reverse[abcd], reverse[dc], to find them in the words, the words stored in the words with index is fastest to find. $O(n)$. Note that when considering suffixes, we explicitly leave out the empty string to avoid counting duplicates. That is, if a palindrome can be created by appending an entire other word to the current word, then we will already consider such a palindrome when considering the empty string as prefix for the other word.

```

1  class Solution(object):
2      def palindromePairs(self, words):
3          # 0 means the word is not reversed, 1 means the
4          # word is reversed
5          words, length, result = sorted([(w, 0, i, len(w))])
6          for i, w in enumerate(words)] +
7              [(w[::-1], 1, i, len(w))]
8          for i, w in enumerate(words)], len(words) * 2, []
9
10         #after the sorting, the same string were nearby, one
11         #is 0 and one is 1
12         for i, (word1, rev1, ind1, len1) in enumerate(words):
13             for j in xrange(i + 1, length):
14                 word2, rev2, ind2, _ = words[j]
15                 #print word1, word2
16                 if word2.startswith(word1): # word2 might
17                     be longer

```

```

13             if ind1 != ind2 and rev1 ^ rev2: # one
14     is reversed one is not
15         rest = word2[len1:]
16         if rest == rest[::-1]: result += ([
17             ind1, ind2],) if rev2 else ([ind2, ind1],) # if rev2 is
18             reversed, the from ind1 to ind2
19             else:
20                 break # from the point of view, break
21             is powerful, this way, we only deal with possible
22             reversed,
23             return result
24

```

There are several other data structures, like balanced trees and hash tables, which give us the possibility to search for a word in a dataset of strings. Then why do we need trie? Although hash table has $O(1)$ time complexity for looking for a key, it is not efficient in the following operations :

- Finding all keys with a common prefix.
- Enumerating a dataset of strings in lexicographical order.

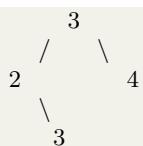
Sorting Lexicographic sorting of a set of keys can be accomplished by building a trie from them, and traversing it in pre-order, printing only the leaves' values. This algorithm is a form of radix sort. This is why it is also called radix tree.

27.4 Bonus

Solve Duplicate Problem in BST When there are duplicates, things can be more complicated, and the college algorithm book did not really tell us what to do when there are duplicates. If you use the definition "left \leq root $<$ right" and you have a tree like:



then adding a "3" duplicate key to this tree will result in:

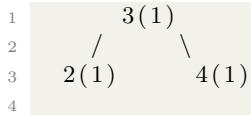


Note that the duplicates are not in contiguous levels.

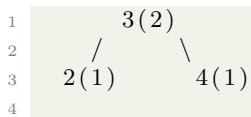
This is a big issue when allowing duplicates in a BST representation as the one above: duplicates may be separated by any number of levels,

so checking for duplicate's existence is not that simple as just checking for immediate children of a node.

An option to avoid this issue is to not represent duplicates structurally (as separate nodes) but instead use a counter that counts the number of occurrences of the key. The previous example would then have a tree like:



and after insertion of the duplicate "3" key it will become:



This simplifies SEARCH, DELETE and INSERT operations, at the expense of some extra bytes and counter operations. In the following content, we assume using definition three so that our BST will have no duplicates.

27.5 LeetCode Problems

1. 144. Binary Tree Preorder Traversal
2. 94. Binary Tree Inorder Traversal
3. 145. Binary Tree Postorder Traversal
4. 589. N-ary Tree Preorder Traversal
5. 590. N-ary Tree Postorder Traversal
6. 429. N-ary Tree Level Order Traversal
7. 103. Binary Tree Zigzag Level Order Traversal(medium)
8. 105. Construct Binary Tree from Preorder and Inorder Traversal

938. Range Sum of BST (Medium)

Given the root node of a **binary search tree**, return the sum of values of all nodes with value between L and R (inclusive).

The binary search tree is guaranteed to have unique values.

```

1 Example 1:
2
3 Input: root = [10,5,15,3,7,null,18], L = 7, R = 15
4 Output: 32
5
6 Example 2:
7
8 Input: root = [10,5,15,3,7,13,18,1,null,6], L = 6, R = 10
9 Output: 23

```

Tree Traversal+Divide and Conquer. We need at most $O(n)$ time complexity. For each node, there are three cases: 1) $L \leq val \leq R$, 2) $val < L$, 3) $val > R$. For the first case it needs to obtain results for both its subtrees and merge with its own val. For the others two, because of the property of BST, only the result of one subtree is needed.

```

1 def rangeSumBST(self, root, L, R):
2     if not root:
3         return 0
4     if L <= root.val <= R:
5         return self.rangeSumBST(root.left, L, R) + self.
rangeSumBST(root.right, L, R) + root.val
6     elif root.val < L: #left is not needed
7         return self.rangeSumBST(root.right, L, R)
8     else: # right subtree is not needed
9         return self.rangeSumBST(root.left, L, R)

```

28

Graph Questions (15%)

In this chapter, we will introduce a variety of algorithms for graphs, and summaries different type of questions from the LeetCode. There are mainly three sections, searching algorithms in graph, which we already introduced in Chapter XX, algorithms that can be applied in the graph include breadth-first search, depth-first search and the topological sort. The second is shortest paths searching algorithms. So for the graph data structure, we usually need to search. Basic DFS/BFS can be applied into any graph data structures. The following sections include more advanced problems, including the concept in Chapter ??.

28.1 Basic BFS and DFS

There are two types of questions :

- that explicitly telling us we need to find a path/shorest/longest path in the graph,
- that implicitly requires us to use DFS/BFS, these type of problems we need to build the graph by ourselves first.

28.1.1 Explicit BFS/DFS

28.1.2 Implicit BFS/DFS

28.1 582. Kill Process (**medium**). Given n processes, each process has a unique PID (process id) and its PPID (parent process id).

Each process only has one parent process, but may have one or more children processes. This is just like a tree structure. Only one process

has PPID that is 0, which means this process has no parent process. All the PIDs will be distinct positive integers.

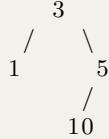
We use two list of integers to represent a list of processes, where the first list contains PID for each process and the second list contains the corresponding PPID.

Now given the two lists, and a PID representing a process you want to kill, return a list of PIDs of processes that will be killed in the end. You should assume that when a process is killed, all its children processes will be killed. No order is required for the final answer.

Example 1:

```
Input:
pid = [1, 3, 10, 5]
ppid = [3, 0, 5, 3]
kill = 5
Output: [5,10]
```

Explanation:



Kill 5 will also kill 10.

Analysis: We know the parent and the child node is a tree-like data structure, which is also a graph. Instead of building a tree data structure first, we use graph defined as defaultdict indexed by the parent node, and the children nodes is a list. In such a graph, finding the killing process is the same as do a DFS/BFS starting from the kill node, we just save all the passing nodes in the process. Here, we only give the DFS solution.

```
1 from collections import defaultdict
2 def killProcess(self, pid, ppid, kill):
3     """
4         :type pid: List[int]
5         :type ppid: List[int]
6         :type kill: int
7         :rtype: List[int]
8     """
9     # first sorting: nlog n,
10    graph = defaultdict(list)
11    for p_id, id in zip(ppid, pid):
12        graph[p_id].append(id)
13
14    q = [kill]
15    path = set()
16    while q:
17        id = q.pop(0)
18        path.add(id)
```

```

19     for neig in graph[id]:
20         if neig in path:
21             continue
22         q.append(neig)
23     return list(path)

```

28.2 Connected Components

28.2 130. Surrounded Regions(medium). Given a 2D board containing 'X' and 'O' (the letter O), capture all regions surrounded by 'X'. A region is captured by flipping all 'O's into 'X's in that surrounded region. Surrounded regions shouldn't be on the border, which means that any 'O' on the border of the board are not flipped to 'X'. Any 'O' that is not on the border and it is not connected to an 'O' on the border will be flipped to 'X'. Two cells are connected if they are adjacent cells connected horizontally or vertically.

Example :

```
X X X X
X O O X
X X O X
X O X X
```

After running your function , the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

Solution 1: Use DFS and visited matrix. First, this is to do operations either filip 'O' or keep it. If 'O' is at the boarder, and any other 'O' that is connected to the boardary 'O', (the connected components that can be found through DFS) will be kept. The complexity is $O(mn)$, m, n is the rows and columns.

```

1 def solve(self, board):
2     """
3     :type board: List[List[str]]
4     :rtype: void Do not return anything, modify board in-
5     place instead.
6     """
7     if not board:
8         return
9     rows, cols = len(board), len(board[0])
10    if rows == 1 or cols == 1:
11        return
12    if rows == 2 and cols == 2:
13        return

```

```

13 moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]
14 # find all connected components to the edge 0, and mark
15 # them as -1,
16 # then flip all 0s in the other parts
17 # change the -1 to 0s
18 visited = [[False for c in range(cols)] for r in range(
19 rows)]
20 def dfs(x, y): # (x, y) is the edge 0s
21     for dx, dy in moves:
22         nx = x + dx
23         ny = y + dy
24         if nx < 0 or nx >= rows or ny < 0 or ny >= cols
25             continue
26         if board[nx][ny] == 'O' and not visited[nx][ny]:
27             visited[nx][ny] = True
28             dfs(nx, ny)
29 # first and last col
30 for i in range(rows):
31     if board[i][0] == 'O' and not visited[i][0]:
32         visited[i][0] = True
33         dfs(i, 0)
34     if board[i][-1] == 'O' and not visited[i][-1]:
35         visited[i][-1] = True
36         dfs(i, cols-1)
37 # first and last row
38 for j in range(cols):
39     if board[0][j] == 'O' and not visited[0][j]:
40         visited[0][j] = True
41         dfs(0, j)
42     if board[rows-1][j] == 'O' and not visited[rows-1][
43 j]:
44         visited[rows-1][j] = True
45         dfs(rows-1, j)
46     for i in range(rows):
47         for j in range(cols):
48             if board[i][j] == 'O' and not visited[i][j]:
49                 board[i][j] = 'X'

```

Solution 2: mark visited 'O' as '-1' to save space. Instead of using a $O(mn)$ space to track the visited vertices, we can just mark the connected components of the boundary 'O' as '-1' in the DFS process, and then we just need another round to iterate the matrix to flip all the remaining 'O' and flip the '-1' back to 'O'.

```

1 def solve(self, board):
2     if not board:
3         return
4     rows, cols = len(board), len(board[0])
5     if rows == 1 or cols == 1:
6         return
7     if rows == 2 and cols == 2:

```

```

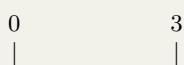
8         return
9     moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]
10    # find all connected components to the edge 0, and mark
11    # them as -1,
12    # then flip all 0s in the other parts
13    # change the -1 to 0s
14    def dfs(x, y): # (x, y) is the edge 0s
15        for dx, dy in moves:
16            nx = x + dx
17            ny = y + dy
18            if nx < 0 or nx >= rows or ny < 0 or ny >= cols
19            :
20                continue
21            if board[nx][ny] == 'O':
22                board[nx][ny] = '-1'
23                dfs(nx, ny)
24
25    # first and last col
26    for i in range(rows):
27        if board[i][0] == 'O':
28            board[i][0] = '-1'
29            dfs(i, 0)
30        if board[i][-1] == 'O' :
31            board[i][-1] = '-1'
32            dfs(i, cols-1)
33    # # first and last row
34    for j in range(cols):
35        if board[0][j] == 'O':
36            board[0][j] = '-1'
37            dfs(0, j)
38        if board[rows-1][j] == 'O':
39            board[rows-1][j] = '-1'
40            dfs(rows-1, j)
41    for i in range(rows):
42        for j in range(cols):
43            if board[i][j] == 'O':
44                board[i][j] = 'X'
45            elif board[i][j] == '-1':
46                board[i][j] = 'O'
47            else:
48                pass

```

28.3 323. Number of Connected Components in an Undirected Graph (medium). Given n nodes labeled from 0 to $n - 1$ and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 1:

Input: $n = 5$ and edges = $[[0, 1], [1, 2], [3, 4]]$



```

1 —— 2      4
Output: 2

Example 2:

Input: n = 5 and edges = [[0, 1], [1, 2], [2, 3], [3, 4]]
0          4
|          |
1 —— 2 —— 3
Output: 1

```

Solution: Use DFS. First, if given n node, and have edges, it will have n components.

```

for n in vertices:
    if n not visited:
        DFS(n) # this is a component traverse its connected
                 components and mark them as visited.

```

Before we start the main part, it is easier if we can convert the edge list into undirected graph using adjacencly list. Because it is undirected, one edge we need to add two directions in the adjancency list.

```

1 def countComponents(self, n, edges):
2     """
3         :type n: int
4         :type edges: List[List[int]]
5         :rtype: int
6     """
7     if not edges:
8         return n
9     def dfs(i):
10        for n in g[i]:
11            if not visited[n]:
12                visited[n] = True
13                dfs(n)
14        return
15    # convert edges into a adjacency list
16    g = [[] for i in range(n)]
17    for i, j in edges:
18        g[i].append(j)
19        g[j].append(i)
20
21    # find components
22    visited = [False]*n
23    ans = 0
24    for i in range(n):
25        if not visited[i]:
26            visited[i] = True
27            dfs(i)
28            ans += 1

```

```
29     return ans
```

28.3 Islands and Bridges

An island is surrounded by water (usually '0's in the matrix) and is formed by connecting adjacent lands horizontally or vertically. An island is actually a definition of the connected components.

1. 463. Island Perimeter
2. 305. Number of Islands II
3. 694. Number of Distinct Islands
4. 711. Number of Distinct Islands II
5. 827. Making A Large Island
6. 695. Max Area of Island
7. 642. Design Search Autocomplete System

28.4 200. Number of Islands. (medium). Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

Input:
11110
11010
11000
00000

Output: 1

Example 2:

Input:
11000
11000
00100
00011

Output: 3

Solution; DFS without extra space.. We use DFS and mark the visited components as '-1' in the grid.

```

1 def numIslands(self, grid):
2     """
3         :type grid: List[List[str]]
4         :rtype: int
5     """
6     if not grid:
7         return 0
8     rows, cols = len(grid), len(grid[0])
9     moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
10    def dfs(x, y):
11        for dx, dy in moves:
12            nx, ny = x + dx, y + dy
13            if nx < 0 or ny < 0 or nx >= rows or ny >= cols
14                continue
15            if grid[nx][ny] == '1':
16                grid[nx][ny] = '-1'
17                dfs(nx, ny)
18        return
19    ans = 0
20    for i in range(rows):
21        for j in range(cols):
22            if grid[i][j] == '1':
23                grid[i][j] = '-1'
24                dfs(i, j)
25            ans += 1
26    return ans

```

28.5 934. Shortest Bridge In a given 2D binary array A, there are two islands. (An island is a 4-directionally connected group of 1s not connected to any other 1s.) Now, we may change 0s to 1s so as to connect the two islands together to form 1 island.

Return the smallest number of 0s that must be flipped. (It is guaranteed that the answer is at least 1.)

Example 1:

Input: [[0, 1], [1, 0]]
Output: 1

Example 2:

Input: [[0, 1, 0], [0, 0, 0], [0, 0, 1]]
Output: 2

Example 3:

Input:
[[1, 1, 1, 1, 1], [1, 0, 0, 0, 1], [1, 0, 1, 0, 1], [1, 0, 0, 0, 1], [1, 1, 1, 1, 1]]
Output: 1

Note :

```
1 <= A.length = A[0].length <= 100
A[i][j] == 0 or A[i][j] == 1
```

Solution 1: DFS to find the complete connected components.

This is a two island problem, First we need to find one node '1' and use DFS to find identify all the '1's compose this first island, in this process, we mark them as '-1'. Then we can do another BFS starts from each node marked as '-1' that is saved in *bfs* to find the shortest path (the first element that is another '1' to make the shortest bridge). A better solution for this is: at each step, we traverse all *bfs* to only expand one step. This is an algorithm that finds the shortest path from multiple starting and multiple ending points. The code is:

```
1 def shortestBridge(self, A):
2     def dfs(i, j):
3         A[i][j] = -1
4         bfs.append((i, j))
5         for x, y in ((i - 1, j), (i + 1, j), (i, j - 1), (i,
6             , j + 1)):
7             if 0 <= x < n and 0 <= y < n and A[x][y] == 1:
8                 dfs(x, y)
9     def first():
10        for i in range(n):
11            for j in range(n):
12                if A[i][j]:
13                    return i, j
14    n, step, bfs = len(A), 0, []
15    dfs(*first())
16    print(A)
17    while bfs:
18        new = []
19        for i, j in bfs:
20            for x, y in ((i - 1, j), (i + 1, j), (i, j - 1),
21                , (i, j + 1)):
22                if 0 <= x < n and 0 <= y < n:
23                    if A[x][y] == 1:
24                        return step
25                    elif not A[x][y]:
26                        A[x][y] = -1
27                        new.append((x, y))
28        step += 1
29        bfs = new
```

28.4 NP-hard Problems

Traveling salesman problems (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. In

fact, there is no polynomial time solution available for this problem as the problem is a known NP-Hard problem.

28.6 943. Find the Shortest Superstring (hard). Given an array A of strings, find any smallest string that contains each string in A as a substring. We may assume that no string in A is substring of another string in A.

Example 1:

Input: ["alex", "loves", "leetcode"]
Output: "alexlovesleetcode"
Explanation: All permutations of "alex", "loves", "leetcode"
would also be accepted.

Example 2:

Input: ["catg", "ctaagt", "gcta", "ttca", "atgcata"]
Output: "gctaagttcatgcata"

Note: $1 \leq A.length \leq 12$, $1 \leq A[i].length \leq 20$ **Solution 1:**
DFS Permutation. First, there are $n!$ possible ways to arrange the strings to connect to get the superstring, and pick the shortest one. This is a typical permutation problems, and when we connect string i to j, we can compute the maximum length of prefix in j that we can skip when connecting. However, with Python, we receive LTE error.

```

1 def shortestSuperstring(self, A):
2     """
3         :type A: List[str]
4         :rtype: str
5     """
6     if not A:
7         return ''
8     n = len(A)
9
10    def getGraph(A):
11        G = [[0 for i in range(n)] for _ in range(n)] #
12        # key is the index, value (index: length of suffix with
13        # the next prefix)
14        if not A:
15            return G
16        for i, s in enumerate(A):
17            for j in range(n):
18                if i == j:
19                    continue
20
21                t = A[j]
22                m = min(len(s), len(t))
23                for l in range(m, 0, -1): #[n, 1]
24                    if s[-l:] == t[0:l]: # suffix and
25                        prefix
26                        G[i][j] = l
27
28    G = getGraph(A)
29    while True:
30        changed = False
31        for i in range(n):
32            for j in range(i+1, n):
33                if G[i][j] > 0:
34                    if G[i][j] < G[j][i]:
35                        G[i][j], G[j][i] = G[j][i], G[i][j]
36                        changed = True
37
38        if not changed:
39            break
40
41    result = A[0]
42    for i in range(1, n):
43        result += A[i][G[i][i]:]
44
45    return result

```

```

24                     break
25             return G
26
27     def dfs(used, d, curr, path, ans, best_path):
28         if curr >= ans[0]:
29             return
30         if d == n:
31             ans[0] = curr
32             best_path[0] = path
33             return
34         for i in range(n):
35             if used & (1<<i):
36                 continue
37             #used[i] = True
38             if curr == 0:
39                 dfs(used|(1<<i), d+1, curr+len(A[i]), path+[i], ans, best_path)
40             else:
41                 dfs(used|(1<<i), d+1, curr+len(A[i])-G[path[-1]][i], path+[i], ans, best_path)
42             #used[i] = False
43     return
44
45
46     G = getGraph(A)
47     ans = [0]
48     for a in A:
49         ans[0] += len(a)
50
51     final_path = [[i for i in range(n)]]
52
53     visited = 0#[False for i in range(n)]
54     dfs(visited, 0, 0, [], ans, final_path)
55
56     # generate result from path
57     final_path = final_path[0]
58     res = A[final_path[0]]
59     for i in range(1, len(final_path)):
60         last = final_path[i-1]
61         cur = final_path[i]
62         l = G[last][cur]
63         res += A[cur][l:]
64     return res

```

Solution 2: Dynamic programming.

29

Dynamic Programming Questions (15%)

In this Chapter, we categorize dynamic programming into three according to the input data types, including Single Sequence (Section 29.1 and Section 29.2), Coordinate (Section 29.4), and Double Sequence(Section 29.5). Each type has its own identifiable characters and can be solved in a certain similar way. In this process, we found the **Forward Induction Method** is the most effective way to identify the recurrence state transfer function. In Forward Induction Method, we start from the base cases (corresponds to the base cases in the DFS solution), and incrementally move to the larger subproblem, and try to induce the state transfer function between current problem and its previous subproblems. If can be induced from only constant subproblems, we have $O(n)$, if relates to all smaller subproblems, we have $O(n^2)$. Using forward inductio method, is intuitive and effective. The only thing we need to note is to try a variety of examples, make sure the recurrence function we found is comprehensive and right. At the end of the section, we would summarize a template for this type of problems solved using dynamic programming. These types include:

1. Single Sequence (50%): This is an easy type too. The states represents if the sequence ends here and include the current element. This way of divide the problem we can obtain the state transfer function easily to find a pattern.
2. Coordinate (15%): 1D or 2D coordinate. This is the easiest type of DP because the state transfer function can be directly obtained through the problem (how to make moves to the next position).
3. Double Sequence (30%): Because double sequence make its state a matrix and subproblem size $O(mn)$, this type of dynamic programming

is similar to coordinate type, within which we just need to figure out the transfer function (moves) ourselves.

The single sequence type dynamic programming is usually applied on the string and array.

Table 29.1: Different Type of Single Sequence Dynamic Programming

Case	Input	Subproblems	$f(n)$	Time	Space
Section 29.1	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n) - > O(1)$
Section 29.2	$O(n)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n)$
Section 29.3	$O(n)$	$O(n^2)$	$O(n)$	$O(n^3)$	$O(n^2)$
Hard	$O(n)$	$O(n^3)$	$O(n)$	$O(n^4)$	$O(n^3)$

Table 29.2: Different Type of Coordinate Dynamic Programming

Case	Input	Subproblems	$f(n)$	Time	Space
Easy	$O(mn)$	$O(mn)$	$O(1)$	$O(mn)$	$O(mn) - > O(m)$
Medium	$O(mn)$	$O(kmn)$	$O(1)$	$O(kmn)$	$O(kmn) - > O(mn)$

Now, let us look at some examples:

29.1 Single Sequence $O(n)$

In this section, we will see how to solve the easy type of dynamic programming shown in Table 29.1, where each subproblem is only dependent on the state of constant number of smaller subproblems. **Subarray** and **Substring** are two types of them. Here, we will see how to using *deduction* method which starts from base case, and gradually get the result of all the cases after it. The examples include problems with one or multiple choice.

Moreover, for this type, because for each subproblem, we only need to look back constant smaller subproblems, we do not even need $O(n)$ space to save all the result, unless you are asked to get the best solution for all subproblems too. Thus, this section generally achieve $O(n)$ and $O(1)$ for time complexity and space complexity, respectively.

1. 276. Paint Fence
2. 256. Paint House
3. 198. House Robber
4. 337. House Robber III (medium)

5. 53. Maximum Subarray (Easy)
6. 152. Maximum Product Subarray
7. 32. Longest Valid Parentheses(hard)

29.1.1 Easy Type

29.1 Paint Fence (L276, *). There is a fence with n posts, each post can be painted with one of the k colors. You have to paint all the posts such that no more than two adjacent fence posts have the same color. Return the total number of ways you can paint the fence. *Note: n and k are non-negative integers.*

Example :

Input: $n = 3$, $k = 2$

Output: 6

Explanation: Take $c1$ as color 1, $c2$ as color 2. All possible ways are:

	post1	post2	post3
1	c1	c1	c2
2	c1	c2	c1
3	c1	c2	c2
4	c2	c1	c1
5	c2	c1	c2
6	c2	c2	c1

Solution: Induction and Multi-choiced State. suppose $n=1$, $dp[1] = k$; when $n=2$, we have two cases: same color with k ways to paint and different color with $k*(k-1)$ ways.

```
dp[1] = k
dp[2] = same + diff; same = k, diff = k*(k-1)
dp[3]: for dp[2].same, we can only have diff colors, diff =
        dp[2].same*(k-1)
        for dp[2].diff, we can have either diff color or
        small color, same = dp[2].diff, diff+=dp[2].diff*(k-1)
```

Thus, using deduction, which is the dynamic programming, the code is:

```
1 def numWays(self, n, k):
2     if n==0 or k==0:
3         return 0
4     if n==1:
5         return k
6
7     same = k
8     diff = k*(k-1)
9     for i in range(3,n+1):
```

```

10     pre_diff = diff
11     diff = (same+diff)*(k-1)
12     same = pre_diff
13     return (same+diff)

```

- 29.2 Paint House (L256, *).** There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a $n \times 3$ cost matrix. For example, $\text{costs}[0][0]$ is the cost of painting house 0 with color red; $\text{costs}[1][2]$ is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Note: All costs are positive integers.

Example :

```

Input: [[17,2,17],[16,16,5],[14,3,19]]
Output: 10
Explanation: Paint house 0 into blue, paint house 1 into
             green, paint house 2 into blue.
             Minimum cost: 2 + 5 + 3 = 10.

```

Solution: Induction and Multi-choiced State. For this problem, each item has three choice, so we need to track the optimal solution for taking each color. $dp[0] = 0$, for one house, return $\min(c1, c2, c3)$.

```

for 1 house: for three choice - (c1, c2, c3), the result is
              min(c1, c2, c3)
for 2 houses: cost of taking c1 = costs [2][c1]+min(dp[1].c2
              , dp[1].c3)
              cost of taking c2 = costs [2][c2]+min(dp[1].c1
              , dp[1].c3)
              cost of taking c3 = costs [2][c3]+min(dp[1].c1
              , dp[1].c2)

```

```

1 def minCost(self, costs):
2     if not costs:
3         return 0
4     c1, c2, c3 = costs[0]
5     n = len(costs)
6     for i in range(1, n):
7         nc1 = costs[i][0] + min(c2, c3)
8         nc2 = costs[i][1] + min(c1, c3)
9         nc3 = costs[i][2] + min(c1, c2)
10        c1, c2, c3 = nc1, nc2, nc3
11    return min(c1, c2, c3)

```

- 29.3 House Robber (L198,*).** You are a professional robber planning to rob houses along a street. Each house has a certain amount of

money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Solution: Induction and Multi-choiced State. For each house has two choice: rob or not rob. Thus the profit for each house can be deducted as follows:

```

1 house: dp[1].rob = p[1], dp[1].not_rob = 0, return max(dp
[1])
2 house: if rob house 2, means we definitely can not rob
house 1. dp[2].rob = dp[1].not_rob + p[2].
           if not rob house 2, means we can choose rob house
1 or not rob house 1. dp[2].not_rob = max(dp[1].rob, dp
[1].not_rob)

```

```

1 def rob(self, nums):
2     if not nums:
3         return 0
4     if len(nums)==1:
5         return nums[0]
6     rob = nums[0]
7     not_rob = 0
8     for i in range(1, len(nums)):
9         new_rob = not_rob + nums[i]
10        new_not_rob = max(rob, not_rob)
11        rob, not_rob = new_rob, new_not_rob
12    return max(rob, not_rob)

```

29.4 House Robber III (L337, medium). The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

Example 1:

Input: [3, 2, 3, null, 3, null, 1]



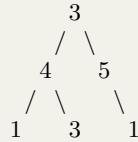
```
3     1
```

Output: 7

Explanation: Maximum amount of money the thief can rob = 3
 $+ 3 + 1 = 7$.

Example 2:

Input: [3, 4, 5, 1, 3, null, 1]



Output: 9

Explanation: Maximum amount of money the thief can rob = 4
 $+ 5 = 9$.

Solution: Induction + Tree Traversal + Multi-choiced State.

This is a dynamic programming applied on tree structure. The brute force still takes $O(2^n)$, where n is the total nodes of the tree. Also, for the tree structure, naturally, the result of a node dependent on the result of its both left and right subtree. When the subtree is empty, then we return (0, 0) for rob and not rob. After we gained the result of left and right subtree each for robbing or not robbing, we merge the result with the current node. Say if we want the result for robbing state for current node: then the left tree and right subtree will only use not robbing, it will be left_not_rob + right_not_rob + current node val. If the current is not robbing, then for the left and right subtree, it both can take rob or not rob state, so we pick the maximum combination of them. Walking through a carefully designed sophisticated enough example is necessary to figure out the process.

```

1 # class TreeNode(object):
2 #     def __init__(self, x):
3 #         self.val = x
4 #         self.left = None
5 #         self.right = None
6 def rob(self, root):
7     def TreeTraversal(root):
8         if not root:
9             return (0, 0)
10
11         l_rob, l_not_rob = TreeTraversal(root.left)
12         r_rob, r_not_rob = TreeTraversal(root.right)
13
14         rob = root.val+(l_not_rob+r_not_rob)
15         not_rob = max(l_rob+r_rob, l_rob+r_not_rob,
16                     l_not_rob+r_not_rob, l_not_rob+r_rob)
  
```

```

16     # not_rob = (max(l_rob , l_not_rob)+max(r_rob ,
17         r_not_rob)
18     return (rob , not_rob)
19     return max(TreeTraversal(root))

```

29.1.2 Subarray Sum: Prefix Sum and Kadane's Algorithm

This subsection is a continuation of the last section. The purpose of separating from the last section is due to the importance of the algorithms—Prefix Sum and Kadane's Algorithms in the problems related to the sum or product of the subarray.

Both Prefix Sum and Kadane's algorithm has used the dynamic programming methodology, and they are highly correlated to each others. They each holds a different perspective to solve a similar problem: one best example is the maximum subarray problem. In the following two sections (Sec 29.1.2 and Sec ??) we will demonstrate how prefix sum is used to solve the maximum subarray problem and how kadane's algorithm which applied dynamic programming directly on this problem. And we show Python code in the next paragraph. After we obtained the prefix sum of the array, using formula $S_{(i,j)} = y_j - y_{i-1}$ can get us the sum of any subarray in the array.

```

1 P = [0]*(len(A)+1)
2 for i, v in enumerate(A):
3     P[i+1] = P[i] + v

```

Prefix Sum and Kadane's Algorithm Application

29.5 Maximum Subarray (L53, *). Given an integer array $nums$, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example :

```

Input : [-2,1,-3,4,-1,2,1,-5,4],
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.

```

Follow up: If you have figured out the $O(n)$ solution, try coding another solution using the divide and conquer approach, which is more subtle.

Solution 1: Prefix Sum. For the maximum subarray problem, we have our answer to be $\max(y_j - y_i)(j > i, j \in [0, n - 1])$, which is equivalent to $\max(y_j - \min(y_i))(i < j), j \in [0, n - 1]$. We can solve the maximum subarray problem using prefix sum with linear $O(n)$ time, where using brute force is $O(n^3)$ and the divide and conquer is $O(nlg n)$. For example, given an array of $[-2, -3, 4, -1, -2, 1, 5, -3]$. We have the following results: The coding:

Table 29.3: Process of using prefix sum for the maximum subarray

Array	-2	-3	4	-1	-2	1	5	3
prefix sum	-2	-5	-1	-2	-4	-3	2	1
Updated prefix sum	-2	-3	4	3	1	2	7	4
current max sum	-2	-2	4	4	4	4	7	7
min prefix sum	-2	-5	-5	-5	-5	-5	-5	-5

```

1 # or we can use import math, math.inf
2
3
4 # Function to compute maximum
5 # subarray sum in linear time.
6 def maximumSumSubarray(nums):
7     if not nums:
8         return 0
9     prefixSum = 0
10    globalA = -sys.maxsize
11    minSub = 0
12    for i in range(len(nums)):
13        prefixSum += nums[i]
14        globalA = max(globalA, prefixSum-minSub)
15        minSub = min(minSub, prefixSum)
16    return globalA
17
18 # Driver Program
19
20 # Test case 1
21 arr1 = [ -2, -3, 4, -1, -2, 1, 5, -3 ]
22 print(maximumSumSubarray(arr1))
23
24 # Test case 2
25 arr2 = [ 4, -8, 9, -4, 1, -8, -1, 6 ]
26 print(maximumSumSubarray(arr2))

```

As we can see, we did not need extra space to save the prefix sum, because each time we only use prefix sum at current index.

Solution 2: Kadane's Algorithm. Another easier perspective using dynamic programming for this problem because we found the key word "Maximum" in the question which is problem that identified in the dynamic programming chapter.

```

dp: the maximum subarray result till index i, which
      includes the current element nums[i]. We need n+1 space
      due to using i-1.
Init: all 0
state transfer function: dp[i] = max(dp[i-1]+nums[i], nums[
      i]); because if for each element, we can either continue
      the previous subarray or start a new subarray.
Request: max(dp)

```

However, to do space optimization, we only need to track the current maximum dp and since $dp[i]$ is only related to $dp[i - 1]$. For the last example, the newly updated prefix sum is $-2, -3, 4, 3, 1, 2, 7, 4$. The comparison result can be seen in Table 29.3.

```

1 def maxSubArray(self, nums):
2     if not nums:
3         return 0
4     dp = [-float('inf')] * (len(nums) + 1)
5     for i, n in enumerate(nums):
6         dp[i+1] = max(dp[i] + n, n)
7     return max(dp)

```

It can also be extended to string, where it is used to compute the number of occurrence of each char in a query substring. For example, 1177. Can Make Palindrome from Substring

```

1 def presum(s):
2     n = len(s)
3     ps = [defaultdict(int)] * (n+1)
4     for i, c in enumerate(s):
5         ps[i+1] = ps[i].copy()
6         ps[i+1][c] += 1
7     return ps
8
9 def helper(queries, ps):
10    ans = []
11    for s, e, k in queries:
12        ct = ps[e+1]
13        num_odd = 0
14        for key in ct.keys():
15            occurrence = ct[key]
16            if key in ps[s]:
17                occurrence -= ps[s][key]

```

Kadane's algorithm Actually, the above simple dynamic programming is exactly the same as an algorithm called Kadane's algorithm, Kadane's algorithm begins with a simple inductive question: if we know the maximum subarray sum ending at position i , what is the maximum subarray sum ending at position $i + 1$? The answer turns out to be relatively straightforward: either the maximum subarray sum ending at position $i + 1$ includes the maximum subarray sum ending at position i as a prefix, or it doesn't. Thus, we can compute the maximum subarray sum ending at position i for all positions i by iterating once over the array. As we go, we simply keep track of the maximum sum we've ever seen. Thus, the problem can be solved with the following code, expressed here in Python:

```

1 def max_subarray(A):
2     max_ending_here = max_so_far = A[0]
3     for x in A[1:]:

```

```

4     max_ending_here = max(x, max_ending_here + x)
5     max_so_far = max(max_so_far, max_ending_here)
6     return max_so_far

```

The algorithm can also be easily modified to keep track of the starting and ending indices of the maximum subarray (when max_so_far changes) as well as the case where we want to allow zero-length subarrays (with implicit sum 0) if all elements are negative.

Because of the way this algorithm uses optimal substructures (the maximum subarray ending at each position is calculated in a simple way from a related but smaller and overlapping subproblem: the maximum subarray ending at the previous position) this algorithm can be viewed as a simple/trivial example of dynamic programming.

Prefix Sum to get BCR convert this problem to best time to buy and sell stock problem. $[0, -2, -1, -4, 0, -1, 1, 2, -3, 1]$, which is to find the maximum benefit, $\Rightarrow O(n)$, use prefix_sum, the difference is we set prefix_sum to 0 when it is smaller than 0, $O(n)$. Or we can try two pointers.

```

1 from sys import maxint
2 def maxSubArray(self, nums):
3     """
4         :type nums: List[int]
5         :rtype: int
6     """
7     max_so_far = -maxint - 1
8     prefix_sum= 0
9     for i in range(0, len(nums)):
10         prefix_sum+= nums[i]
11         if (max_so_far < prefix_sum):
12             max_so_far = prefix_sum
13
14         if prefix_sum< 0:
15             prefix_sum= 0
16     return max_so_far

```

Generalize Kadane's Algorithm

Because we can still do space optimization to the above solution, we use one variable to replace the dp array, and we track the maximum dp in the for loop instead of obtaining the maximum value at the end. Also, if we rename the dp to max_ending_here and the max(dp) to max_so_far, the code is as follows:

```

1 def maximumSumSubarray(arr, n):
2     if not arr:
3         return 0
4     max_ending_here = 0
5     max_so_far = -sys.maxsize
6     for i in range(len(arr)):
7         max_ending_here = max(max_ending_here+arr[i], arr[i])

```

```

8     max_so_far = max(max_so_far, max_ending_here)
9     return max_so_far

```

This space-wise optimized dynamic programming solution to the maximum subarray problem is exactly the Kadane's algorithm. Kadane's algorithm begins with a simple inductive question: if we know the maximum subarray sum ending at position i , what is the maximum subarray sum ending at position $i+1$? The answer turns out to be relatively straightforward: either the maximum subarray sum ending at position $i+1$ includes the maximum subarray sum ending at position i as a prefix, or it doesn't. Thus, we can compute the maximum subarray sum ending at position i for all positions i by iterating once over the array. As we go, we simply keep track of the maximum sum we've ever seen. Thus, the problem can be solved with the following code, expressed here in Python:

```

1 def max_subarray(A):
2     max_ending_here = max_so_far = A[0]
3     for x in A[1:]:
4         max_ending_here = max(x, max_ending_here + x)
5         max_so_far = max(max_so_far, max_ending_here)
6     return max_so_far

```

The algorithm can also be easily modified to keep track of the starting and ending indices of the maximum subarray (when `max_so_far` changes) as well as the case where we want to allow zero-length subarrays (with implicit sum 0) if all elements are negative. For example:

Now, let us see how we do maximum subarray with product operation instead of the sum.

29.6 Maximum Product Subarray (L152, **). Given an integer array `nums`, find the contiguous subarray within an array (containing at least one number) which has the largest product.

Example 1:

```

Input: [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.

```

Example 2:

```

Input: [-2,0,-1]
Output: 0
Explanation: The result cannot be 2, because [-2,-1] is not
a subarray.

```

Solution: Kadane's Algorithm with product. For the product, the difference compared with sum is the `max_ending_here` is not necessarily computed from the previous value with current element; if the element is negative it might even become the smallest. So that

we need to track another variable, the `min_ending_here`. Let us see the Python code which is a straightforward implementation of the product-modified kadane's algorithm.

```

1 from sys import maxsize
2 class Solution(object):
3     def maxProduct(self, nums):
4         """
5             :type nums: List[int]
6             :rtype: int
7         """
8         if not nums:
9             return 0
10        n = len(nums)
11        max_so_far = nums[0]
12        min_local, max_local = nums[0], nums[0]
13        for i in range(1, n):
14            a = min_local*nums[i]
15            b = max_local*nums[i]
16            max_local = max(nums[i], a, b)
17            min_local = min(nums[i], a, b)
18            max_so_far = max(max_so_far, max_local)
19        return max_so_far

```

29.1.3 Subarray or Substring

It will lower the complexity from $O(n^2)$ or $O(n^3)$ to $O(n)$.

29.7 Longest Valid Parentheses (L32, hard). Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

Example 1:

```

Input: "(() "
Output: 2
Explanation: The longest valid parentheses substring is "()
"

```

Example 2:

```

Input: ")()())"
Output: 4
Explanation: The longest valid parentheses substring is "()"

```

Solution 1: Dynamic programming. We define the state to be the longest length ends at this position. We would know only ')' can possibly have value larger than 0. At all position of '(' it is 0. As our define, for the following case:

```

1 ") ( ) ( ) )"

```

```

2 dp 0 0 2 0 4 0
3   ") ( ( ( ) ) ) (" 
4 dp 0 0 2 0 0 0 2 4 8 0

```

Thus, when we are at position ')', we look for $i-1$, there are two cases:

- 1) if $s[i-1] == '('$, it is an closure, $dp[i] += 2$, then we check $dp[i-2]$ to connect with previous longest length. for example in case 1, ")()()", where $dp[i] = 4$.
- 2) if $s[i-1] == ')'$, then we check at position $i-1-dp[i-1]$, in case , at $dp[i] = 8$, if at its corresponding position we check if it is '(' . If it is we increase the count by 2, and connect it with previous position .

```

1 def longestValidParentheses(self, s):
2     """
3         :type s: str
4         :rtype: int
5     """
6     if not s:
7         return 0
8     dp = [0]*len(s)
9     for i in range(1, len(s)):
10        c = s[i]
11        if c == ')':#check previous position
12            if s[i-1] == '(':#this is the closure
13                dp[i] += 2
14                if i-2 >= 0: #connect with previous length
15                    dp[i] += dp[i-2]
16            if s[i-1] == ')': #look at i-1-dp[i-1] for '('
17                if i-1-dp[i-1] >= 0 and s[i-1-dp[i-1]] == '('
18                :
19                    dp[i] = dp[i-1]+2
20                    if i-1-dp[i-1]-1 >= 0: # connect with
21                        previous length
22                            dp[i-1] += dp[i-1-dp[i-1]-1]
23
24    print(dp)
25    return max(dp)
26 # input "((())())()"
27 # output [0, 0, 2, 4, 0, 0, 2, 0, 0]

```

Solution 2: Using Stack.

```

1 def longestValidParentheses(self, s):
2     if not s:
3         return 0
4     stack=[-1]
5     ans = 0
6     for i, c in enumerate(s):
7         if c == '(':
8             stack.append(i)
9         else:
10             if stack:
11                 stack.pop()
12                 if not stack:

```

```

13         stack.append(i)
14     else:
15         ans = max(ans, i - stack[-1])
16 return ans

```

29.1.4 Exercise

1. 639. Decode Ways II (hard)

29.2 Single Sequence $O(n^2)$

In this section, we will analysis the second type in Table 29.1 where we have $O(n)$ subproblems, and each subproblem is dependent on all the previous smaller subproblems, thus gave us $O(n^2)$ time complexity. The problems here further can be categorized as **Subsequence** and **Splitting**.

1. 300. Longest Increasing Subsequence (medium)
2. 139. Word Break (Medium)
3. 132. Palindrome Partitioning II (hard)
4. 123. Best Time to Buy and Sell Stock III (hard)
5. 818. Race Car (hard)

29.2.1 Subsequence

29.8 Longest Increasing Subsequence (L300, medium). Given an unsorted array of integers, find the length of longest increasing subsequence.

Example :

Input : [10, 9, 2, 5, 3, 7, 101, 18]

Output : 4

Explanation: The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4.

Note: (1) There may be more than one LIS combination, it is only necessary for you to return the length. (2) Your algorithm should run in $O(n^2)$ complexity.

Follow up: Could you improve it to $O(n \log n)$ time complexity?

Solution 1: Induction. For each subproblem, we show the result as follows. Each state $dp[i]$ we represents the longest increasing subsequence ends with $nums[i]$. The reconstruction depends on all the previous $i-1$ subproblems, as shown in Eq. 29.1.

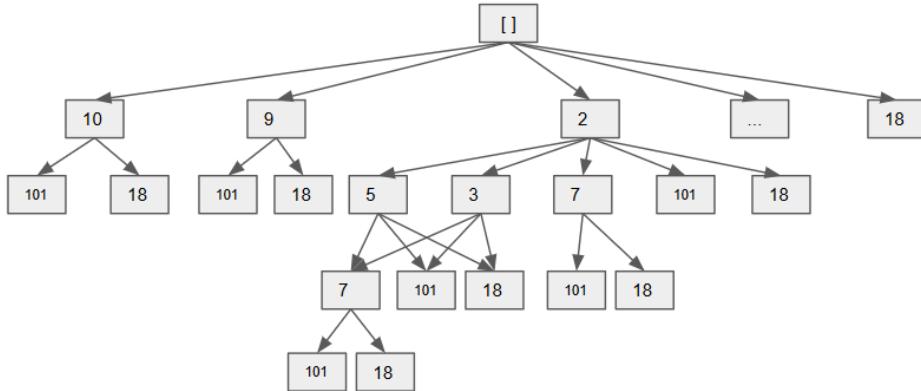


Figure 29.1: State Transfer Tree Structure for LIS, each path represents a possible solution. Each arrow represents a move: find an element in the following elements that's larger than the current node.

```

1 subproblem: [], [10], [10, 9],
              [10, 9, 2], [10, 9, 2, 5], [10, 9, 2, 5, 3], [10, 9, 2, 5, 3, 7]...
2 Choice:
3 ans:      0,   1,   1,   1,   2,
            3,
```

$$f(i) = \begin{cases} 1 + \max(f(j)), & 0 < j < i, arr[j] < arr[i]; \\ 1 & \text{otherwise} \end{cases} \quad (29.1)$$

```

1 class Solution(object):
2     def lengthOfLIS(self, nums):
3         """
4             :type nums: List[int]
5             :rtype: int
6         """
7         max_count = 0
8         LIS = [0]*(len(nums)+1) # the LIS for array ends
9         with index i
10            for i in range(len(nums)): # start with 10
11                max_before = 0
12                for j in range(i):
13                    if nums[i] > nums[j]:
14                        max_before = max(max_before, LIS[j+1])
15                LIS[i+1] = max_before+1
16        return max(LIS)
```

29.2.2 Splitting

Need to figure out how to fill out the two-dimensional dp matrix for splitting.

29.9 Word Break (L139, **). Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, determine if

s can be segmented into a space-separated sequence of one or more dictionary words. Note: (1) The same word in the dictionary may be reused multiple times in the segmentation. (2) You may assume the dictionary does not contain duplicate words.

Example 1:

Input: $s = \text{"leetcode"}$, $\text{wordDict} = [\text{"leet"}, \text{"code"}]$
 Output: true
 Explanation: Return true because "leetcode" can be segmented as "leet code".

Example 2:

Input: $s = \text{"applepenapple"}$, $\text{wordDict} = [\text{"apple"}, \text{"pen"}]$
 Output: true
 Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
 Note that you are allowed to reuse a dictionary word.

Example 3:

Input: $s = \text{"catsandog"}$, $\text{wordDict} = [\text{"cats"}, \text{"dog"}, \text{"sand"}, \text{"and"}, \text{"cat"}]$
 Output: false

Solution: Induction + Splitting. Like most of single sequence problem, we have n overlapping subproblems, for example of "leetcode".

subproblem: ' ', 'l', 'le', 'lee', 'leet', 'leetc', 'leetco ', 'leetcod', 'leetcode'.	ans: 1, 0, 0, 0, 1, 0, 0, 0, 1
--	--

Thus, deduction still works here. We manually write down the result of each subproblem. Suppose we are trying to achieve answer for 'leet', how does it work? if 'lee' is true and 't' is true, then we have true. Or, if 'le' is true, and 'et' is true, we have true. unlike problems before, the ans for 'leet' can only be constructed from all the previous smaller problems.

```

1 def wordBreak(self, s, wordDict):
2     wordDict = set(wordDict)
3     n = len(s)
4     dp = [False]*(n+1)
5     dp[0] = True #set 1 for empty str ''
6     for i in range(1, n+1):
7         for j in range(i):
8             if dp[j] and s[j:i] in wordDict: # check
9                 previous result, and new word s[j:i]
10                dp[i] = True

```

```

10
11     return dp[-1]

```

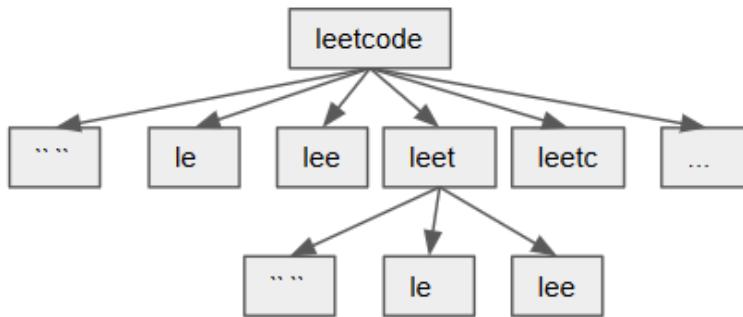


Figure 29.2: Word Break with DFS. For the tree, each arrow means check the word = parent-child and then recursively check the result of child.

DFS+Memo. To understand why each subproblem depends on $O(n)$ even smaller subproblem, we can look at the process solving the problem with DFS shown in Fig. 29.2 (we can also draw a tree structure which will be more obvious). For “leetcode” and “leet” they both computed the subproblem ”, ’l’, ’le’, ’lee’. Thus we can use memory to save solved problems. From the tree structure, for each root node, it has $O(n)$ subbranches. So we should see why. To complete this, we give the code for the DFS version.

```

1 def wordBreak( self , s , wordDict ) :
2     wordDict = set(wordDict)
3     #backtracking
4     def DFS( start , end , memo ) :
5         if start >= end :
6             return True
7         if start not in memo :
8             if s[ start :end ] in wordDict :
9                 memo[ start ] = True

```

```

10         return memo[start]
11
12     for i in range(start, end+1):
13         word = s[start:i] #i is the splitting point
14         if word in wordDict:
15             if i not in memo:
16                 memo[i] = DFS(i, end, memo)
17             if memo[i]:
18                 return True
19         memo[start] = False
20
21     return memo[start]
22
23 return DFS(0, n, {})

```

- 29.10 Palindrome Partitioning II (L132, ***)** Given a string s, partition s such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of s.

Example:

```

Input: "aab"
Output: 1
Explanation: The palindrome partitioning ["aa","b"] could
be produced using 1 cut.

```

Solution: use two dp. one to track if it is pal and the other is to compute the cuts.

```

1 def minCut(self, s):
2     """
3         :type s: str
4         :rtype: int
5     """
6     pal = [[False for _ in range(len(s))] for _ in
7            range(len(s)) ]
7     cuts = [len(s)-i-1 for i in range(len(s)) ]
8     for start in range(len(s)-1,-1,-1):
9         for end in range(start, len(s)):
10            if s[start] == s[end] and (end-start < 2 or
11                pal[start+1][end-1]):
12                pal[start][end] = True
13                if end == len(s)-1:
14                    cuts[start] = 0
15                else:
16                    cuts[start] = min(cuts[start], 1+
17                        cuts[end+1])
16    return cuts[0]

```

- 29.11 Best Time to Buy and Sell Stock III (L123, hard).** Say you have an array for which the ith element is the price of a given stock on day i. Design an algorithm to find the maximum profit. You may

complete at most two transactions. *Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).*

Example 1:

```
Input: [3,3,5,0,0,3,1,4]
Output: 6
Explanation: Buy on day 4 (price = 0) and sell on day 6 (
    price = 3), profit = 3-0 = 3.
        Then buy on day 7 (price = 1) and sell on day
        8 (price = 4), profit = 4-1 = 3.
```

Example 2:

```
\begin{lstlisting}
Input: [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (
    price = 5), profit = 5-1 = 4.
        Note that you cannot buy on day 1, buy on day
        2 and sell them later, as you are
            engaging multiple transactions at the same
            time. You must sell before buying again.
```

Example 3:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max
    profit = 0.
```

Solution: the difference compared with I is that we need at most two times of transaction. We split the array into two parts from i, the max profit we can get till i and the max profit we can get from i to n. To get the maximum profit of each part is the same as the problem I. At last, the answer is $\text{maxpreProfit}[i] + \text{postProfit}[i], (0 \leq i \leq n - 1)$. However, we would get $O(n^2)$ time complexity if we use the following code, it has a lot of redundancy.

```
1 from sys import maxsize
2 class Solution:
3     def maxProfit(self, prices):
4         """
5             :type prices: List[int]
6             :rtype: int
7         """
8     def maxProfitI(start, end):
9
10        if start == end:
11            return 0
12        max_global_profit = 0
13        min_local = prices[start]
14        for i in range(start+1, end+1):
```

```

15             max_global_profit= max(max_global_profit ,
16     prices [ i]-min_local)
17             min_local = min(min_local ,  prices [ i])
18             return  max_global_profit
19
20     if  not  prices:
21         return  0
22     n = len(prices)
23     min_local = prices [ 0]
24     preProfit ,  postProfit = [ 0]*n ,  [ 0]*n
25
26     for  i  in  range(n):
27         preProfit [ i] = maxProfitI(0 , i)
28         postProfit [ i] = maxProfitI(i , n-1)
29     maxProfit = max([ pre+post  for  pre ,  post  in  zip(
            preProfit ,  postProfit)])
            return  maxProfit

```

To avoid repeat work, we can use a for loop to get all the value of preProfit, and use another to get values for postProfit. For the post-Profit, we need to traverse from the end to the start of the array in reverse direction, this way we track the local_max and the profit is going to be local_max - prices[i], and both keep a global max profit. The code is as follows:

```

1 def  maxProfit(self ,  prices):
2     """
3     :type  prices:  List[int]
4     :rtype:  int
5     """
6
7     if  not  prices:
8         return  0
9     n = len(prices)
10
11    preProfit ,  postProfit = [ 0]*n ,  [ 0]*n
12    #get  preProfit ,  from 0-n ,  track  the  mini_local ,
13    #global_max
14    min_local = prices [ 0]
15    max_global_profit = 0
16    for  i  in  range(1 , n):
17        max_global_profit= max(max_global_profit ,  prices [ i
18        ]-min_local)
19        min_local = min(min_local ,  prices [ i])
20        preProfit [ i] = max_global_profit
21    #get  postProfit ,  from  n-1  to  0 ,  track  the  max_local ,
22    #global_min
23    max_local = prices [-1]
24    max_global_profit = 0
25    for  i  in  range(n-1 , -1 , -1):
26        max_global_profit= max(max_global_profit ,  max_local
27        -prices [ i])
28        max_local = max(max_local ,  prices [ i])
29        postProfit [ i] = max_global_profit

```

```

25     # iterate preProfit and postProfit to get the maximum
26     profit
27     maxProfit = max([pre+post for pre, post in zip(
28         preProfit, postProfit)])
29     return maxProfit

```

818. Race Car (hard)

29.3 Single Sequence $O(n^3)$

The difference of this type of single sequence is that there are not only n subproblems for a sequence of size n , each subarray is $A[0:i]$, $i=[0, n]$. There will be n^2 subproblems, each states as subarray $A[i : j]$, $i \leq j$. Usually for this type, it shows such optimal substructure $dp[i][j] = f(dp[i][k], dp[k][j])$, $k \in [i, j]$. This would give us the $O(n^3)$ time complexity and $O(n^2)$ space complexity. The classical examples of this type of problem is matrix-multiplication as explained in *Introduction to Algorithms* and stone game.

29.3.1 Interval

Problems include Stone Game, Burst Ballons, and Scramble String. The features of this type of dynamic programming is we try to get the min/max/count of a range of array; and the state transfer function updates through the range by from the big range to small rang.

29.12 486. Predict the Winner (medium) Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by the player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player. This continues until all the scores have been chosen. The player with the maximum score wins.

Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.

Example 1: Input: [1, 5, 2]. Output: False

Explanation: Initially, player 1 can choose between 1 and 2.

If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If player 2 chooses 5, then player 1 will be left with 1 (or 2). So, final score of player 1 is $1 + 2 = 3$, and player 2 is 5. Hence, player 1 will never be the winner and you need to return False.

Example 2: Input: [1, 5, 233, 7]. Output: True

Explanation: Player 1 first chooses 1. Then player 2 have to choose between 5 and 7. No matter which number player 2 choose, player 1 can choose 233. Finally, player 1 has more score (234) than player 2 (12), so you need to return True representing player1 can win.

Note:

1. $1 \leq \text{length of the array} \leq 20$.
2. Any scores in the given array are non-negative integers and will not exceed 10,000,000.
3. If the scores of both players are equal, then player 1 is still the winner.

Solution: At first, we can not use $f[i]$ to denote the state, because we can choose element from both the left and the right side, we use $f[i][j]$ instead, which represents the maximum value we can get from i to j range. Second, when we deal with problem with potential accumulate value, we can use $\text{sum}[i][j]$ to represent the sum in the range $i - j$. Each player take actions to maximize their total points, $f[i][j]$, it has two choice: left, right, which left $f[i+1][j]$ and $f[i][j-1]$ respectively for player two to choose. In order to gain the maximum scores in range $[i,j]$ we need to optimize it by making sure $f[i+1][j]$ and $f[i][j-1]$ we choose the minimum value from. Therefore, we have state transfer function: $f[i][j] = \text{sum}[i][j] - \min(f[i+1][j], f[i][j-1])$. Each subproblem relies on only two subproblems, which makes the total time complexity $O(n^2)$. This is actually a game theory type. According to the function: if the range is 1, when $i == j$, the value is $\text{nums}[i]$, which is the initialization. For the loop, the first for loop is the range: from size 2 to n , the second for loop to get the start index i in range $[0, n - l]$, then the end index $j = i + l - 1$. The answer for this problem is: if $f[0][-1] \geq \text{sum}/2$. If it is, then it is true.

The process of the for loop is we initialize the diagonal element, and fill out element on the right upper side, which is upper diagonal.

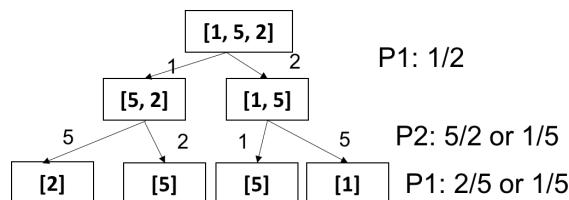


Figure 29.3: Caption

```
1 def PredictTheWinner(nums):
```

```

2     """
3     :type nums: List[int]
4     :rtype: bool
5     """
6     if not nums:
7         return False
8     if len(nums) == 1:
9         return True
10    #sum[i, j] = sum[j+1] - sum[i]
11    sums = nums[:]
12    for i in range(1, len(nums)):
13        sums[i] += sums[i-1]
14    sums.insert(0, 0)
15
16    dp = [[0 for col in range(len(nums))] for row in
17          range(len(nums)) ]
18    for i in range(len(nums)):
19        dp[i][i] = nums[i]
20
21    for l in range(2, len(nums)+1):
22        for i in range(0, len(nums)-l+1): #start 0, end
23            len_l = len(nums)
24            j = i+l-1
25            dp[i][j] = (sums[j+1]-sums[i]) - min(dp[i+1][j],
26                                         dp[i][j-1])
27            n = len(nums)
28            return dp[0][n-1] >= sums[-1]/2

```

Else, we use $f[i][j] = \max(\text{nums}[i] - f[i+1][j], \text{nums}[j] - f[i][j-1])$ to represent the difference of the points gained by player one compared with player two. When $f[i][j]$ is the state of player one, then $f[i][j-1]$ and $f[i+1][j]$ are the potential states of player two.

```

1 class Solution:
2     def PredictTheWinner(self, nums):
3         """
4             :type nums: List[int]
5             :rtype: bool
6         """
7         n = len(nums)
8         if n == 1 or n%2==0 : return True
9         dp = [[0]*n for _ in range(n)]
10        for l in range(2, len(nums)+1):
11            for i in range(0, len(nums)-l+1): #start 0, end
12                len_l = len(nums)
13                j = i+l-1
14                dp[i][j] = max(nums[j] - dp[i][j-1], nums[i] -
15                               dp[i+1][j])
16                return dp[0][-1] >= 0

```

Actually the for loop we can use a simpler one. However, it is harder to understand to code compared with the standard version.

```

1 for i in range(n-1, -1, -1):

```

```

2         dp[i][i] = nums[i] #initialization
3         for j in range(i+1,n):
4             dp[i][j] = max(nums[j] - dp[i][j-1], nums[i]
- dp[i+1][j])

```

29.13 Stone Game

There is a stone game. At the beginning of the game the player picks n piles of stones in a line. The goal is to merge the stones in one pile observing the following rules:

At each step of the game, the player can merge two adjacent piles to a new pile. The score is the number of stones in the new pile. You are to determine the minimum of the total score. Example For [4, 1, 1, 4], in the best solution, the total score is 18:

Merge second and third piles [4, 2, 4], score +2 Merge the first two piles [6, 4], score +6 Merge the last two piles [10], score +10

Other two examples: [1, 1, 1, 1] return 8 [4, 4, 5, 9] return 43

29.14 312. Burst Balloons

Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If you burst balloon i you will get $\text{nums}[left] * \text{nums}[i] * \text{nums}[right]$ coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note: (1) You may imagine $\text{nums}[-1] = \text{nums}[n] = 1$. They are not real therefore you can not burst them. (2) $0 \leq \text{nums}[i] \leq 500$, $0 \leq \text{nums}[i] \leq 100$

Example :

Given [3, 1, 5, 8]

Return 167

nums = [3, 1, 5, 8] —>	[3, 5, 8] —>	[3, 8] —>	[8] —>	[]
coins = 3*1*5	+ 3*5*8	+ 1*3*8	+ 1*8*1	
= 167				

at first burst c[i][k-1] then burst c[k+1][j], then burst k,

```

1 class Solution:
2     def maxCoins(self, nums):
3         """
4             :type nums: List[int]
5             :rtype: int
6         """

```

```

7     n = len(nums)
8     nums.insert(0,1)
9     nums.append(1)
10
11    c = [[0 for _ in range(n+2)] for _ in range(n+2)]
12    for l in range(1, n+1): #length [1,n]
13        for i in range(1, n-l+2): #start [1, n-l+1]
14            j = i+l-1 #end =i+l-1
15
16            #function is a k for loop
17            for k in range(i, j+1):
18                c[i][j] = max(c[i][j], c[i][k-1]+nums[i
19 -1]*nums[k]*nums[j+1]+c[k+1][j])
20
21    #return from 1 to n
22    return c[1][n]

```

29.15 516. Longest Palindromic Subsequence

Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.

Example 1:

```

Input :
"bbbab"
Output :
4
One possible longest palindromic subsequence is "bbbb".

```

Example 2:

```

Input :
"cbbd"
Output :
2
One possible longest palindromic subsequence is "bb".

```

Solution: for this problem, we have state $dp[i][j]$ means from i to j, the length of the longest palindromic subsequence. $dp[i][i] = 1$. Then we use this range to fill in the dp matrix (upper triangle.)

```

1 def longestPalindromeSubseq(self, s):
2     """
3         :type s: str
4         :rtype: int
5     """
6     nums=s
7     if not nums:
8         return 0
9     if len(nums)==1:
10        return 1
11
12    def isPanlidrome(s):
13        l, r= 0, len(s)-1
14        while l<=r:

```

```

15         if s[1]!=s[r]:
16             return False
17         else:
18             l+=1
19             r-=1
20         return True
21
22     if isPanlidrome(s): #to speed up
23         return len(s)
24
25     rows=len(nums)
26     dp=[[0 for col in range(rows)] for row in range(
27     rows)]
28     for i in range(0,rows):
29         dp[i][i] = 1
30
31     for l in range(2, rows+1): #use a length
32         for i in range(0,rows-l+1): #start 0, end len -
33             j = i+l-1
34             if j>rows:
35                 continue
36             if s[i]==s[j]:
37                 dp[i][j] = dp[i+1][j-1]+2
38             else:
39                 left_size ,right_size = dp[i][j-1],dp[i
40                 +1][j]
41                 dp[i][j]= max(dp[i][j-1], right_size)
42     print(dp)
43     return dp[0][rows-1]

```

Or else, we can say, i need to be from $i+1$ to i , from big to small, j need to from $j-1$ or j to j , from small to big.

```

1 for (int i = n - 1; i >= 0; --i) {
2     dp[i][i] = 1;
3     for (int j = i + 1; j < n; ++j) {
4         if (s[i] == s[j]) {
5             dp[i][j] = dp[i + 1][j - 1] + 2;
6         } else {
7             dp[i][j] = max(dp[i + 1][j], dp[i][j -
8                 1]);
9         }
10    }
}

```

Now to do the space optimization:

```

1 class Solution {
2 public:
3     int longestPalindromSubseq(string s) {
4         int n = s.size(), res = 0;
5         vector<int> dp(n, 1);
6         for (int i = n - 1; i >= 0; --i) {

```

```

7     int len = 0;
8     for (int j = i + 1; j < n; ++j) {
9         int t = dp[j];
10        if (s[i] == s[j]) {
11            dp[j] = len + 2;
12        }
13        len = max(len, t);
14    }
15    for (int num : dp) res = max(res, num);
16    return res;
17}
18}
19};

```

29.4 Coordinate: BFS and DP

In this type of problems, we are given an array or a matrix with 1D or 2D axis. We either do 'optimization' to find the minimum path sum, or do the 'counting' to get the total number of paths, or check if we can start from A and end at B.

Two-dimensional. For a $O(mn)$ sized coordinate, Tab. 29.4 shows two different types: one there will only be $O(mn)$, and the other is $O(kmn)$, k here normally represents number of steps. Because a 2D coordinate is inherently a graph, so this type is closely related to the graph traversal algorithms; BFS for counting and DFS for the optimization problems. For this

Table 29.4: Different Type of Coordinate Dynamic Programming

Case	Input	Subproblems	$f(n)$	Time	Space
Easy	$O(mn)$	$O(mn)$	$O(1)$	$O(mn)$	$O(mn) - > O(m)$
Medium	$O(mn)$	$O(kmn)$	$O(1)$	$O(kmn)$	$O(kmn) - > O(mn)$

type of problems, understanding the BFS related solution is more important than just memorizing the template of the dynamic programming solution. There, we will use two sections: Counting: BFS and DP in Sec. 29.4.1 and Optimization in Sec. ?? with LeetCode examples to learn how to solve this type of dynamic programming problems.

29.4.1 One Time Traversal

In this section, we want to explore how we can modify our solution from BFS to the dynamic programming. Inherently, dynamic programming solutions for this type of problems are the optimized Breath-first-search.

Counting

In this type, any location in the coordinate will be only visted once. Thus, it gives $O(mn)$ time complexity.

62. Unique Paths

```

1 A robot is located at the top-left corner of a m x n grid (
2   marked 'Start' in the diagram below).
3 The robot can only move either down or right at any point in
4   time. The robot is trying to reach the bottom-right corner of
5   the grid (marked 'Finish' in the diagram below).
6 How many possible unique paths are there?
7 Above is a 3 x 7 grid. How many possible unique paths are there?
8 Note: m and n will be at most 100.
9
10 Example 1:
11
12
13 Input: m = 3, n = 2
14 Output: 3
15 Explanation:
16 From the top-left corner, there are a total of 3 ways to reach
17   the bottom-right corner:
18 1. Right -> Right -> Down
19 2. Right -> Down -> Right
20 3. Down -> Right -> Right
21 Example 2:
22
23 Input: m = 7, n = 3
24 Output: 28

```

BFS. Fig. 29.4 shows the BFS traversal process in the matrix. We can clearly see that each node and edge is only visited once. The BFS solution is straightforward and is the best solution. We use bfs to track the nodes in the queue at each level, and dp to record the unique paths to location (i, j) . Because each location is only visted once, thus, at each level, using the same dp will have no conflict.

```

1 # BFS
2 def uniquePaths(self, m, n):
3     dp = [[0 for _ in range(n)] for _ in range(m)]
4     dp[0][0] = 1
5     bfs = set([(0,0)])
6     dirs = [(1, 0), (0,1)]
7     while bfs:
8         new_bfs = set()
9         for x, y in bfs:
10            for dx, dy in dirs:
11                nx, ny = x+dx, y+dy
12                if 0<=nx< m and 0<=ny< n:

```

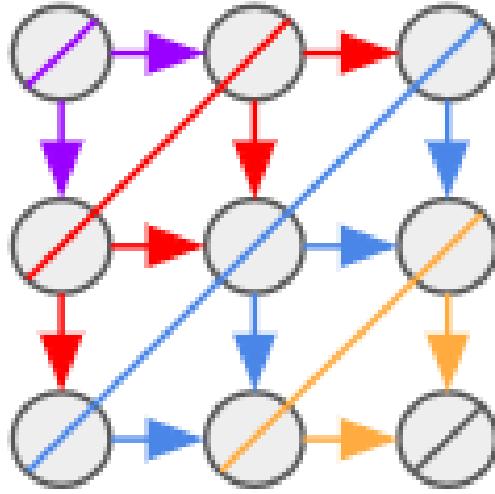


Figure 29.4: One Time Graph Traversal. Different color means different levels of traversal.

```

13         dp[nx][ny] += dp[x][y]
14         new_bfs.add((nx, ny))
15     bfs = new_bfs
16     return dp[m-1][n-1]

```

Dynamic Programming. In the BFS solution, we use a set to track the nodes at each level. However, its corresponding dynamic programming solution should design a way to obtain the result of current state only dependent on the previous computed state. Here each position has a different state: (x, y) and $f[x][y]$ denotes the number of unique paths from start position $(0, 0)$ to (x, y) . The state transfer function: $f[x][y] = f[x - 1][y] + f[x][y - 1]$. If we initialize the boundary locations (the first row and the first column), and we visit each location by loop over row and col, then we can get the dynamic programming solution.

```

1 def uniquePaths(self, m, n):
2     if m==0 or n==0:
3         return 0
4     dp=[[0 for col in range(n)] for row in range(m)]
5     dp[0][0]=1
6     #initialize row 0
7     for col in range(1,n):
8         dp[0][col] = dp[0][col-1]
9     #initialize col 0
10    for row in range(1,m):
11        dp[row][0] = dp[row-1][0]
12
13    for row in range(1,m):
14        for col in range(1,n):

```

```

15     dp [ row ] [ col ] = dp [ row - 1 ] [ col ] + dp [ row ] [ col - 1 ]
16     return dp [ m - 1 ] [ n - 1 ]

```

377. Combination Sum IV (medium)

```

1 Given an integer array with all positive numbers and no
2 duplicates, find the number of possible combinations that add
3 up to a positive integer target.
4
5 Example:
6
7 nums = [1, 2, 3]
8 target = 4
9
10 The possible combination ways are:
11 (1, 1, 1, 1)
12 (1, 1, 2)
13 (1, 2, 1)
14 (1, 3)
15 (2, 1, 1)
16 (2, 2)
17 (3, 1)
18
19 Note that different sequences are counted as different
20 combinations.
21
22 Therefore the output is 7.
23
24 Follow up:
25 What if negative numbers are allowed in the given array?
26 How does it change the problem?
27 What limitation we need to add to the question to allow negative
28 numbers?

```

Target as Climbing Stairs. Analysis: The DFS+MEMO solution is given in Section ???. However, because we just need to count the number, which makes the dynamic programming possible. From the DFS solution, we can see the state depends on the target, thus we can define a dp array that use ($\text{target}+1$) space. This is like climbing stairs, each time we can either go 1, or 2, or 3 steps.

```

1 [1, 2, 3], t = 4
2 t = 0: dp[0] = 1, []
3 t = 1: t(0)+1, dp[1] = 1; [1]
4 t = 2: t(0)+2, dp[2] = 1, t(1)+1, dp[2]+=1, dp[2] = 2; [2], [1,
5   1]
6 t = 3: t(0)+3, dp[3]=1, t(1)+2, dp[3]+=1, t(2)+1, dp[3]+=2, dp
7   [3] = 4, [3], [1, 2], [2, 1], [1, 1, 1]
8 t = 4: t(0)+4, dp[4]=0, t(1)+3, dp[4]+=1, t(2)+2, dp[4]+=2, t(3)
9   +1, dp[4]+=4, dp[4] = 7, [1, 3], [2, 2], [1, 1, 2], [3, 1],
10  [1, 2, 1], [2, 1, 1], [1, 1, 1, 1]

```

```

1 def combinationSum4(self, nums, target):
2   """

```

```

3   :type nums: List[int]
4   :type target: int
5   :rtype: int
6   """
7   nums.sort()
8   n = len(nums)
9   dp = [0]*(target+1)
10  dp[0] = 1
11  for t in range(1, target+1):
12      for n in nums:
13          if t-n >= 0:
14              dp[t] += dp[t-n]
15          else:
16              break
17  return dp[-1]

```

Optimization

64. Minimum Path Sum (medium)

```

1 Given a m x n grid filled with non-negative numbers, find a path
2 from top left to bottom right which minimizes the sum of all
3 numbers along its path.
4 Note: You can only move either down or right at any point in
5 time.
6 Example 1:
7
8 [[1,3,1],
9  [1,5,1],
10 [4,2,1]]

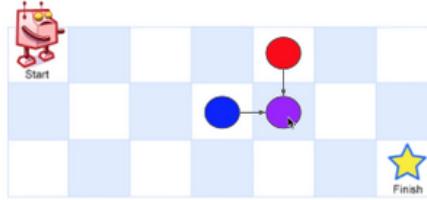
```

Given the above grid map, return 7. Because the path $1 \rightarrow 3 \rightarrow 1 \rightarrow 1 \rightarrow 1$ minimizes the sum. **Dynamic Programming**. For this problem, it is exactly the same as all the previous problems, the only difference is the state transfer function. $f(i, j) = g(i, j) + \min(f(i - 1, j), f(i, j - 1))$.

```

1 # dynamic programming
2 def minPathSum(self, grid):
3     if not grid:
4         return 0
5     rows, cols = len(grid), len(grid[0])
6     dp = [[0 for _ in range(cols)] for _ in range(rows)]
7     dp[0][0] = grid[0][0]
8
9     # initialize row
10    for c in range(1, cols):
11        dp[0][c] = dp[0][c-1] + grid[0][c]
12
13    # initialize col
14    for r in range(1, rows):

```



$$\text{Purple circle} = \min(\text{Red circle}, \text{Blue circle}) + \text{grid}[y][x]$$

$$\begin{array}{c} \text{Red circle} \\ \text{Start} \end{array} = \text{grid}[0][0]$$

Figure 29.5: Caption

```

15     dp[r][0] = dp[r-1][0] + grid[r][0]
16
17     for r in range(1, rows):
18         for c in range(1, cols):
19             dp[r][c] = grid[r][c] + min(dp[r-1][c], dp[r][c-1])
20     return dp[-1][-1]

```

Dynamic Programming with Space Optimization. As can be seen, each time when we update $sum[i][j]$, we only need $sum[i-1][j]$ (at the current column) and $sum[i][j-1]$ (at the left column). So we need not maintain the full $m*n$ matrix. Maintaining two columns is enough and now we have the following code.

```

1 rows, cols= len(grid), len(grid[0])
2     #O(rows)
3     pre, cur =[0]*rows, [0]*rows
4     #initialize the first col, walk from the (0,0) ->(1,0)
5     ->(row,0)
6     pre[0]=grid[0][0]
7
8     for row in range(1, rows):
9         pre[row]=pre[row-1]+grid[row][0] #this is equal to
10        cost[0][row]
11        for col in range(1, cols):
12            cur[0] = pre[0]+grid[0][col] #initialize the first
13            row, current [0][0]
14            for row in range(1, rows):
15                cur[row]= min(cur[row-1], pre[row])+grid[row][
16                col]
17            pre, cur = cur, pre
18        return pre[rows-1]

```

Further inspecting the above code, it can be seen that maintaining pre is for recovering $pre[i]$, which is simply $cur[i]$ before its update. So it is enough to use only one vector. Now the space is further optimized and the

code also gets shorter.

```

1 rows, cols= len(grid),len(grid[0])
2     #O(rows)
3     cur = [0]*rows
4     #intialize the the first col, walk from the (0,0)->(1,0)
->(row,0)
5     cur[0]=grid[0][0]
6     for row in range(1, rows):
7         cur[row]=cur[row-1]+grid[row][0]
8     for col in range(1, cols):
9         cur[0] = cur[0]+grid[0][col]  #intialize the first
row
10    for row in range(1, rows):
11        cur[row]= min(cur[row-1], cur[row])+grid[row][
col]
12
13    return cur[rows-1]
```

Now, we use $O(1)$ space by reusing the original grid.

```

1 rows, cols= len(grid),len(grid[0])
2     #O(1) space by reusing the space here
3     for i in range(0, rows):
4         for j in range(0, cols):
5             if i==0 and j ==0:
6                 continue
7             elif i==0 :
8                 grid[i][j]+=grid[i][j-1]
9             elif j==0:
10                 grid[i][j]+=grid[i-1][j]
11             else:
12                 grid[i][j]+= min(grid[i-1][j], grid[i][j-1])
13
14    return grid[rows-1][cols-1]
```

29.4.2 Multiple-time Traversal

In this type, we need to traverse each location for K times, making K steps of moves thus we can get the final solution. This will have $O(kmn)$ time complexity.

Two-dimensional Coordinate

935. Knight Dialer (Medium)

- 1 A chess knight can move as indicated in the chess diagram below:
- 2 This time, we place our chess knight on any numbered key of a phone pad (indicated above), and the knight makes $N-1$ hops. Each hop must be from one key to another numbered key.
- 3
- 4 Each time it lands on a key (including the initial placement of the knight), it presses the number of that key, pressing N digits total.
- 5

1	2	3
4	5	6
7	8	9
0		

Figure 29.6: Caption

```

6 How many distinct numbers can you dial in this manner?
7
8 Since the answer may be large , output the answer modulo  $10^9 +$ 
9
10 Example 1:
11
12 Input: 1
13 Output: 10
14
15 Example 2:
16
17 Input: 2
18 Output: 20
19
20 Example 3:
21
22 Input: 3
23 Output: 46
24
25 Note:
26
27     1 <= N <= 5000

```

Most Naive BFS. Analysis: First, we need to figure out from each number, where is the possible next moves. We would have get this dictionary: $moves = \{0 : [4, 6], 1 : [6, 8], 2 : [7, 9], 3 : [4, 8], 4 : [0, 3, 9], 5 : [], 6 : [0, 1, 7], 7 : [2, 6], 8 : [1, 3], 9 : [2, 4]\}$. This is not exactly a coordinate, however, because we can make endless move, we would have a graph. The brute force is we put $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$ as the start positions, and we use BFS to control the steps, the total number of paths is the sum over of all the leaves. At each step, we would do two things 1) generate a list to save all the possible next numbers; 2) if it reaches to the leaves, sum up all the nodes.

```

1 # naive BFS solution
2 def knightDialer(self , N):
3     """
4         :type N: int
5         :rtype: int
6     """
7     if N == 1:
8         return 10

```

```

9 moves = {0:[4, 6], 1:[6, 8], 2: [7, 9], 3: [4,8], 4: [0, 3,
9], 5:[], 6:[0,1,7], 7:[2,6], 8:[1,3],9:[2,4]} #4, 6 has
three
10 bfs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] # all starting points
11 step = 1
12 while bfs:
13     new = []
14     for i in bfs:
15         new += moves[i]
16     step += 1
17     bfs = new
18     if step == N:
19         return len(bfs)% (10**9+7)
20

```

Optimized BFS. However, the brute force BFS only passed 18/120 test cases. To improve it further, we know that we only need a counter to record the counter of each number in that level. This way, bfs is replaced with a counter. Now, the new code is:

```

1 #optimized BFS exactly a DP
2 def knightDialer(self, N):
3     MOD = 10**9+7
4     if N == 1:
5         return 10
6     moves = {0:[4, 6], 1:[6, 8], 2: [7, 9], 3: [4,8], 4: [0, 3,
9], 5:[], 6:[0,1,7], 7:[2,6], 8:[1,3],9:[2,4]} #4, 6 has
three
7     bfs = [1]*10
8     step = 1
9
10    while bfs:
11        size = 0
12        new = [0]*10
13        for idx, count in enumerate(bfs):
14            for m in moves[idx]:
15                new[m] += count
16                new[m] %= MOD
17        step += 1
18        bfs = new
19        if step == N:
20            return sum(bfs)% (MOD)

```

Optimized Dynamic Programming. This is exactly a dynamic programming algorithm: $new[m] += bfs[i]$, for example, from 1 we can move to 6,8,so that we have $f(1,n) = f(6,n - 1) + f(8,n - 1)$. So here a state is represented by $bfs[num]$ and $step$, and it saves the count at each state. Now, we write it in the way of dp template:

```

1 # optimized dynamic programming template
2 def knightDialer(self, N):
3     MOD = 10**9+7

```

```

4     moves = {0:[4, 6], 1:[6, 8], 2: [7, 9], 3: [4,8], 4: [0, 3,
9], 5:[], 6:[0,1,7], 7:[2,6], 8:[1,3],9:[2,4]} #4, 6 has
      three
5     dp = [1]*10
6
7     for step in range(N-1):
8         size = 0
9         new_dp = [0]*10
10        for idx, count in enumerate(dp):
11            for m in moves[idx]:
12                new_dp[m] += count
13                new_dp[m] %= MOD
14        dp = new_dp
15
16
17    return sum(dp)%MOD

```

688. Knight Probability in Chessboard (Medium)

- 1 On an NxN chessboard, a knight starts at the r-th row and c-th column and attempts to make exactly K moves. The rows and columns are 0 indexed, so the top-left square is (0, 0), and the bottom-right square is (N-1, N-1).
- 2 A chess knight has 8 possible moves it can make, as illustrated below. Each move is two squares in a cardinal direction, then one square in an orthogonal direction.
- 3 Each time the knight is to move, it chooses one of eight possible moves uniformly at random (even if the piece would go off the chessboard) and moves there.
- 4 The knight continues moving until it has made exactly K moves or has moved off the chessboard. Return the probability that the knight remains on the board after it has stopped moving.
- 5 Example:
- 6
- 7 Input: 3, 2, 0, 0
- 8 Output: 0.0625
- 9 Explanation: There are two moves (to (1,2), (2,1)) that will keep the knight on the board.
- 10 From each of those positions, there are also two moves that will keep the knight on the board.
- 11 The total probability the knight stays on the board is 0.0625.

Optimized BFS. Analysis: Each time we can make 8 moves, thus after K steps, we can have 8^K total unique paths. Thus, we just need to get the total number of paths that it ends within the board (valid paths). The first step is to write down the possible moves or directions. And, then we initialize a two-dimensional array dp to record the number of paths end at (i, j) after k steps. Using a BFS solution, each time we just need to save all the unique positions can be reached at that step.

```

1 # Optimized BFS solution
2 def knightProbability(self , N, K, r , c):
3     dirs = [[-2, -1], [-2, 1], [-1, -2], [-1, 2], [1, -2], [1,
4         2], [2, -1],[2, 1]]
5     dp = [[0 for _ in range(N)] for _ in range(N) ]
6     total = 8**K
7     last_pos = set([(r , c)])
8     dp[r][c]=1
9
10    for step in range(K):
11        new_pos = set()
12        new_dp = [[0 for _ in range(N)] for _ in range(N) ]
13        for x, y in last_pos:
14            for dx, dy in dirs:
15                nx = x+dx
16                ny = y+dy
17                if 0<=nx<N and 0<=ny<N:
18                    new_dp[nx][ny] += dp[x][y]
19                    new_pos.add((nx, ny))
20        last_pos = new_pos
21        dp = new_dp
22
23    return float(sum(map(sum, dp)))/total

```

Optimized Dynamic Programming. If we delete the last_pos and directly use the dp to use as a way to visit the last positions, this is a space optimized dynamic programming solution. And this solution is nearly the fastest; compared with the above solution, each step we cut down the cost of maintain a set (a hashmap) dynamically.

```

1 # Best Dynamic Programming Solution
2 def knightProbability(self , N, K, r , c):
3     dirs = [[-2, -1], [-2, 1], [-1, -2], [-1, 2], [1, -2], [1,
4         2], [2, -1],[2, 1]]
5     dp = [[0 for _ in range(N)] for _ in range(N) ]
6     total = 8**K
7     dp[r][c]=1
8
9     for step in range(K):
10        new_dp = [[0 for _ in range(N)] for _ in range(N) ]
11        for i in range(N):
12            for j in range(N):
13                if dp[i][j] == 0:
14                    continue #not available position
15                for dx, dy in dirs:
16                    nx, ny = i+dx, j+dy
17                    if 0<=nx<N and 0<=ny<N:
18                        new_dp[nx][ny] += dp[i][j]
19        dp = new_dp
20
21    return float(sum(map(sum, dp)))/total

```

One-dimensional Coordinate

For one-dimensional, it is the same as two-dimensional, and it could be even simpler.

70. Climbing Stairs (Easy)

```

1 You are climbing a stair case. It takes n steps to reach to the
2 top .
3 Each time you can either climb 1 or 2 steps . In how many
4 distinct ways can you climb to the top ?
5 Note: Given n will be a positive integer .
6
7 Example 1:
8
9 Input: 2
10 Output: 2
11 Explanation: There are two ways to climb to the top .
12 1. 1 step + 1 step
13 2. 2 steps
14
15 Example 2:
16
17 Input: 3
18 Output: 3
19 Explanation: There are three ways to climb to the top .
20 1. 1 step + 1 step + 1 step
21 2. 1 step + 2 steps
22 3. 2 steps + 1 step

```

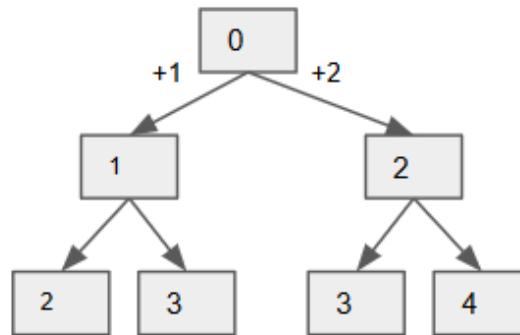


Figure 29.7: Tree Structure for One dimensional coordinate

BFS. Fig 29.7 demonstrates the state transfer relation between different position. First, we can solve it using our standard BFS. In this problem, we do not know the level of the tree structure, so the end condition is while the bfs is not empty. Thus, eventually the bfs is set to empty, and the result of the dp is empty too. So we use a global variable *ans* to track our result.

```

1 # BFS
2 def climbStairs(self, n):
3     dp = [0]*(n+1)
4     dp[0] = 1 # init starting point 0 to 1
5     dirs = [1, 2]
6     bfs = set([0])
7     ans = 0
8     while bfs:
9         new_dp = [0]*(n+1)
10        new_bfs = set()
11        for i in bfs: #pos
12            for dx in dirs:
13                nx = i+dx
14                if 0 <= nx <= n:
15                    new_dp[nx] += dp[i]
16                    new_bfs.add(nx)
17        ans += dp[-1]
18        bfs, dp = new_bfs, new_dp
19    return ans

```

Dynamic Programming. If we observe the tree structure, we have the state transfer function $f(i) = f(i - 1) + f(i - 2)$. Thus, a single for loop starts from 2, the dp list can be filled in without overlap.

```

1 # Dynamic Programming
2 def climbStairs(self, n):
3     dp = [0]*(n+1)
4     dp[0] = 1 # init starting point 0 to 1
5     dp[1] = 1
6
7     for i in range(2, n+1):
8         dp[i] = dp[i-1] + dp[i-2]
9     return dp[-1]

```

The BFS and the Dynamic Programming has the same time and space complexity.

29.4.3 Generalization

1. State: $f[x]$ or $f[x][y]$ to denote the optimum value or count, or check the workability of whole solutions till axis x for 1D and (x, y) for 2D;
2. Function: usually for $f[x]$, we connect $f[x]Rf[x - 1]$, or $f[x][y]Rf[x - 1][y], f[x][y - 1]$;
3. Initialization: for $f[x]$ we initialize the starting point, sometimes we need extra 1 space, with size $n + 1$; for $f[x][y]$ we need to initialize elements from row 0 and col 0;
4. Answer: Usually it is $f[n - 1]$ or $f[m - 1][n - 1]$;

Space Optimization For $f[i] = \max(f[i - 1], f[i - 2] + A[i])$, it can be converted into $f[i \% 2] = \max(f[(i - 1)\%2], f[(i - 2)\%2])$. Also, we can

directly using the original matrix or array to save the state results. Note: there are possible ways to optimize the space complexity, we can do it from $O(m * n)$ to $O(m + n)$ to $O(1)$ which we get by reusing the original grid or array.

One-Time Traversal

```

1 dp[ i ][ j ] := answer of A[0->i][0->j]
2
3 #template
4 dp[n+1][m+1]
5 for i in range(n):
6     for j in range(m):
7         dp[ i ][ j ] = f(dp[ pre_i ][ pre_j ])
8 return f(dp)

```

Multiple-Dimensional Traversal

```

1 dp[ k ][ i ][ j ] := answer of A[0->i][0->j] after k steps
2
3 #template
4 dp[ k ][ n+1 ][ m+1 ]
5 for _ in range(k):
6     for i in range(n):
7         for j in range(m):
8             dp[ k ][ i ][ j ] = f(dp[ k-1 ][ pre_i ][ pre_j ])
9 return f(dp)

```

29.5 Double Sequence: Pattern Matching DP

	<i>aa</i>	A	B	D
<i>aa</i>	0	0	0	0
A	0	1	1	1
B	0	1	2	2
C	0	1	2	2
D	0	1	2	3

Figure 29.8: Longest Common Subsequence

In this section, we focus on double sequence P and S with input size $O(m) + O(n)$. Because double sequence can naturally be arranged to be a matrix with size $(m+1) \times (n+1)$. Here we have extra row and extra column,

it happens because we put empty char " at the beginning of each string to better initialize and get result even for empty string too. One example is shown in Fig. 29.8. This mostly make the time complexity for this section $O(mn)$. This type of dynamic programming can be generalized to coordinate problems. The difference is the moves are not given as in coordinate section (29.4.1).

We need to find the deduction rules or say recurrence relation ourselves. Most of the time, the moves are around their neighbors: for (i, j) , we have potential positions of $(i-1, j-1)$, $(i-1, j)$, $(i, j-1)$. For example, in the case of Longest Common Subsequence in Fig. 29.8, if current $P[i]$ and $S[j]$ matches, then it only depends on $dp[i-1][j-1]$. If not, it depends on the relation between $(P(0, i), S(0, j-1))$ and $(P(0, i-1), S(0, j))$. *Filling out an exemplary table manually can guide us find the rules.* If we do so, we would find out that even problems marked as hard from LeetCode is solvable.

Brute Force. For the brute force solution: we need t

Problems shown in this section include:

1. 72. Edit Distance
2. 712. Minimum ASCII Delete Sum for Two Strings
3. 115. Distinct Subsequences (hard)
4. 44. Wildcard Matching (hard)

29.5.1 Longest Common Subsequence

Problem Definition: Given two string A and B, for example A is "ABCD", and B is "ABD", the longest common subsequence is "ABD", so the length of the longest common subsequence is 3.

Coordinate+Moves. Because each has m and n subproblems, two sequence make it a matrix problem. The result of the above example is shown in Fig. 29.8. We can try to observe the problem and generalize the moves or state transfer function. For the red marked positions, the char in string A and B are the same. So, the length would be the result of its previous substrings plus one. Otherwise as the black marked positions, it is the maximum of the left and above positions. And the math equation is shown in Eq. 29.2. To initialize, we need to initialize the first row and the first column, which is $f[i][0] = 0, f[0][j] = 0$.

$$f[i][j] = \begin{cases} 1 + f[i-1][j-1], & a[i-1] == b[j-1]; \\ \max(f[i-1][j], f[i][j-1]) & \text{otherwise} \end{cases} \quad (29.2)$$

The Python code is shown as follow:

```

1 def LCSLen(S1, S2):
2     if not S1 or not S2:
3         return 0
4     n, m = len(S1), len(S2)
5     f = [[0]*(m+1) for _ in range(n+1)]
6     #init f[0][0] = 0
7     for i in range(n):
8         for j in range(m):
9             f[i+1][j+1] = f[i][j]+1 if S1[i]==S2[j] else max(f[i]
10                ][j+1], f[i+1][j])
11     print(f)
12     return f[-1][-1]
13 S1 = "ABCD"
14 S2 = "ABD"
15 LCSLen(S1, S2)
16 # output
17 # [[0, 0, 0, 0], [0, 1, 1, 1], [0, 1, 2, 2], [0, 1, 2, 2], [0,
18   1, 2, 3]]
# 3

```

29.5.2 Other Problems

There are more pattern matching related dynamic programming, we give them in this section.

29.16 72. Edit Distance (hard). Given two words word1 and word2, find the minimum number of operations required to convert word1 to word2. You have the following 3 operations permitted on a word: Insert a character, Delete a character, Replace a character.

Example 1:

```

Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')

```

Example 2:

```

Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')

```

Coordinate+Deduction. This is similar to the LCS length. We use $f[i][j]$ to denote the minimum number of operations needed to make

the previous i chars in S_1 to be the same as the first j chars in S_2 . The upbound of the minimum edit distance is $\max(m,n)$ by replacing and insertion. The most important step is to decide the transfer function: to get the result of current state $f[i][j]$. If directly filling in the matrix is obscure, then we can try the recursive:

```

DFS("horse", "rose")
= DFS("hors", "ros") # no edit at e
= DFS("hor", "ro") # no edit at s
= 1 + min(DFS("ho", "ro"), # delete "r" from longer one
          DFS("hor", "r"), # insert "o" at the longer one, left
          "hor" and "r" to match
          DFS("ho", "r")), # replace "r" in the longer one with
          "o" in the shorter one, left "ho" and "r" to match

```

Be written as equation 29.3. Thus, it can be solved by dynamic programming.

$$f[i][j] = \begin{cases} \min(f[i][j-1], f[i-1][j], f[i-1][j-1]) + 1, & S_1[i-1] \neq S_1[j-1]; \\ f[i-1][j-1] & \text{otherwise} \end{cases} \quad (29.3)$$

The Python code is as follows:

```

1 def minDistance(word1, word2):
2     if not word1:
3         if not word2:
4             return 0
5         else:
6             return len(word2)
7     if not word2:
8         return len(word1)
9     dp = [[0 for col in range(len(word2)+1)] for row in
10           range(len(word1)+1)]
11     rows=len(word1)
12     cols=len(word2)
13     for row in range(1, rows+1):
14         dp[row][0] = row
15     for col in range(1, cols+1):
16         dp[0][col] = col
17
18     for i in range(1, rows+1):
19         for j in range(1, cols+1):
20             if word1[i-1]==word2[j-1]:
21                 dp[i][j]=dp[i-1][j-1]
22             else:
23                 dp[i][j]=min(dp[i-1][j]+1, dp[i][j-1]+1, dp
24 [i-1][j-1]+1) # add, delete, replace
25     return dp[rows][cols]

```

29.17 115. Distinct Subsequences (hard).

Given a string S and a string T , count the number of distinct subsequences of S which equals T .

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Example 1:

Input: S = "rabbbit", T = "rabbit"

Output: 3

Explanation:

As shown below, there are 3 ways you can generate "rabbit" from S.

(The caret symbol \wedge means the chosen letters)

```
rabbbit
^~~~ ~~
```

```
rabbbit
~~ ~~~~
```

```
rabbbit
~~~ ~~~
```

Example 2:

Input: S = "babgbag", T = "bag"

Output: 5

Explanation:

As shown below, there are 5 ways you can generate "bag" from S.

(The caret symbol \wedge means the chosen letters)

```
babgbag
~~ ~
```

```
babgbag
~~ ~
```

```
babgbag
~ ~~~
```

```
babgbag
~ ~~~
```

```
babgbag
~~~
```

Coordinate. Here still we need to fill out a matrix. We would see if the length of s is smaller than the length of t: then it is 0. If the length is equal, which is the diagonal in the matrix, then it only depends on position (i-1, j-1) and s(i), s(j). For the lower part of the matrix it has different rule: for example, s = 'ab', t = 'a', because s[i] != t[j], then we need to find s[0, i-1] with t[0, j]. if it equals, we can check the dp[i-1][j-1].

	'	a	b	a
''	1	0	0	0

a	1	1	0	0
b	1	1	1	0
b	1	1	2	0
a	1	2	2	2

```

1 def numDistinct(self, s, t):
2     if not s or not t:
3         if not s and t:
4             return 0
5         else:
6             return 1
7
8     rows, cols = len(s), len(t)
9     if cols > rows:
10        return 0
11    if cols == rows:
12        return 1 if s==t else 0
13
14    # initialize
15    dp = [[0 for c in range(cols+1)] for r in range(rows+1)]
16    for r in range(rows):
17        dp[r+1][0] = 1
18    dp[0][0] = 1
19
20    # fill out the lower part
21    for i in range(rows):
22        for j in range(min(i+1,cols)):
23            if i==j: # diagonal
24                if s[i] == t[j]:
25                    dp[i+1][j+1] = dp[i][j]
26                else: # lower half of the matrix
27                    if s[i] == t[j]:
28                        dp[i+1][j+1] = dp[i][j+1]+dp[i][j] # dp
29                        [i][j] is because they equal, so check previous i,j,
30                        else:
31                            dp[i+1][j+1] = dp[i][j+1] # check the
subsequence before this char in S is the same as t
return dp[-1][-1]
```

29.18 44. Wildcard Matching (hard). Given an input string (*s*) and a pattern (*p*), implement wildcard pattern matching with support for '?' and '*'.

'?' Matches any single character. '*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

Note:

s could be empty and contains only lowercase letters a-z. *p* could be empty and contains only lowercase letters a-z, and characters like ? or *.

Example 1:

```
Input:
s = "aa"
p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

Example 2:

```
Input:
s = "aa"
p = "*"
Output: true
Explanation: '*' matches any sequence.
```

Example 3:

```
Input:
s = "cb"
p = "?a"
Output: false
Explanation: '?' matches 'c', but the second letter is 'a',
which does not match 'b'.
```

Example 4:

```
Input:
s = "adceb"
p = "*a*b"
Output: true
Explanation: The first '*' matches the empty sequence,
while the second '*' matches the substring "dce".
```

Solution 1: Complete Search: DFS. We start from the first element in s and p with index i, j. If it is a '?', or $s[i]=p[j]$, we match $\text{dfs}(i+1, j+1)$. The more complex one is for '*', it can go from empty to full length of s. Therefore, we call $\text{dfs}(k, j+1)$, $k \in [i, n]$. Check if any of these recursive calls return True. It receives LTE error.

```
1 def isMatch(self, s, p):
2     """
3         :type s: str
4         :type p: str
5         :rtype: bool
6     """
7     ns, np = len(s), len(p)
8     def helper(si, pi):
9         if si == ns and pi == np:
10             return True
11         elif si == ns or pi == np:
12             if si == ns: # if pattern left, make sure its
all '*'!
```

```

13         for i in range(pi, np):
14             if p[i] != '*':
15                 return False
16         return True
17     else: # if string left, return False
18         return False
19
20     if p[pi] in ['?', '*']:
21         if p[pi] == '?':
22             return helper(si+1, pi+1)
23         else:
24             for i in range(si, ns+1): # we can match
all till the end
25             #print(i)
26             if helper(i, pi+1):
27                 return True
28             return False
29     else:
30         if p[pi] != s[si]:
31             return False
32         return helper(si+1, pi+1)
33
34 return helper(0, 0)

```

Solution 2: Dynamic programming. Same as all the above problems, we try to fill out the dp table ourselves. If it is a '?', check $dp[i-1][j-1]$, if $p[i]==s[j]$, check $dp[i-1][j-1]$. For '*', if it is treated as ", check $dp[i-1][j]$ (above), because it can be any length of string, we check left $dp[i][j-1]$.

	'	a	d	c	e	b
'	1	0	0	0	0	0
*	1	1	1	1	1	1
a	0	1	0	0	0	0
*	0	1	1	1	1	1
b	0	0	0	0	0	1
*	0	0	0	0	0	1

```

1 def isMatch(self, s, p):
2     ns, np = len(s), len(p)
3     dp = [[False for c in range(ns+1)] for r in range(np+1)]
4
5     # initialize
6     dp[0][0] = True
7     for r in range(1, np+1):
8         if p[r-1] == '*' and dp[r-1][0]:
9             dp[r][0] = True
10
11    # dp main
12    for r in range(1, np+1):
13        for c in range(1, ns+1):
14            if p[r-1] == '?':
15                dp[r][c] = dp[r-1][c-1]

```

```

16         elif p[r-1] == '*':
17             dp[r][c] = dp[r-1][c] or dp[r][c-1] # above
18             or left
19             if dp[r][c]:
20                 for nc in range(c+1, ns+1):
21                     dp[r][nc] = True
22                     break
23             else:
24                 if dp[r-1][c-1] and p[r-1] == s[c-1]:
25                     dp[r][c] = True
26
27     return dp[-1][-1]

```

29.5.3 Summary

The four elements include:

1. state: $f[i][j]$: i denotes the previous i number of numbers or characters in the first string, j is the previous j elements for the second string; We need to assign $n + 1$ and $m + 1$ for each dimension;
2. function: $f[i][j]$ research how to match the ith element in the first string with the jth element in the second string;
3. initialize: $f[i][0]$ for the first column and $f[0][j]$ for the first row;
4. answer: $f[n][m]$

29.6 Knapsack

The problems in this section are defined as: Given n items with Cost C_i and value V_i , we can choose i items that either 1) equals to an amount S or 2) is bounded by an amount S . We would be required to obtain either 1) maximum values or 2) minimum items. Depends on if we can use one item multiple times, we have three categorizes:

1. 0-1 Knapsack (Section 29.6.1): each item is only allowed to use 0 or 1 time.
2. Unbounded Knapsack(Section 29.6.2): each item is allowed to use unlimited times.
3. Bounded Knapsack(Section 29.6.3): each item is allowed to use a fixed number of times.

How to solve the above three types of questions will be explained and the Python example will be given in the next three subsections (Section 29.6.1,

29.6.2, and 29.6.3) with the second type of restriction that the total cost is bounded by an amount S .

The problems itself is a combination problem with restriction, therefore we can definitely use DFS as the naive solution. Moreover, the problems are not about to simply enumerate all the combinations, its an optimization problems, this is the difference of with memoization to solve these problems. Thus, dynamic programming is not our only choice. We can refer to Section ?? and Section ?? for the DFS based solution and reasoning.

LeetCode problems:

1. 322. Coin Change (**)
 - unbounded, fixed amount.

29.6.1 0-1 Knapsack

In this subsection, each item is only allowed to be used at most one time. This is a combination problem with restriction (total cost be bounded by a given cost or say the total weights of items need to be \leq the capacity of the knapsack).

Given the following example: we can get the maximum value to be 9 by choosing item 3 and 4 each with cost 2.

```
c = [1,1,2,2]
v = [1,2,4,5]
C = 4
```

Solution 1: Combination with DFS. Clearly this is a combination problem, here we give the naive DFS solution. The time complexity if $O(2^n)$.

```
1 def knapsack01DFS(c, v, C):
2     def dfs(s, cur_c, cur_v, ans):
3         ans[0] = max(ans[0], cur_v)
4         if s == n: return
5         for i in range(s, n):
6             if cur_c + c[i] <= C: # restriction
7                 dfs(i + 1, cur_c + c[i], cur_v + v[i], ans)
8
9     ans = [0]
10    n = len(c)
11    dfs(0, 0, 0, ans)
12    return ans[0]
13
14 c = [1,1,2,2]
15 v = [1,2,4,5]
16 C = 4
17 print(knapsack01DFS(c, v, C))
18 # output
19 # 9
```

Solution 2: DFS+MEMO. However, because this is an optimization problem **Solution 3: Dynamic Programming.** Here, we can try to make it iterative with dynamic programming. Here, because we have two variables to track (need modification), we use $dp[i][c]$ to denote maximum

value we can gain with subproblems $(0,i)$ and a cost of c . Thus, the size of the dp matrix is $n \times (C + 1)$. This makes the time complexity of $O(n \times C)$. Like any coordinate type of dynamic programming problems, We definitely need to iterate through two for loops, one for i and the other for c , which one is inside or outside does not matter here. The state transfer function will be: the maximum value of 1) not choose this item, 2) choose this item, which will add $v[i]$ to the value of the first $i-1$ items with cost of $c-c[i]$. $dp[i][c] = \max(dp[i-1][c], dp[i-1][c-c[i]] + v[i])$.

```

1 def knapsack01DP(c, v, C):
2     dp = [[0 for _ in range(C+1)] for r in range(len(c)+1)]
3     for i in range(len(c)):
4         for w in range(c[i], C+1):
5             dp[i+1][w] = max(dp[i][w], dp[i][w-c[i]]+v[i])
6     return dp[-1][-1]

```

Optimize Space. Because when we are updating dp, we use the left upper row to update the right lower row, we can reduce the space to $O(C)$. If we keep the same code as above just with one dimensional dp, then for the later part of updating it is using the updated result from the same level, thus resulting using each item multiple times which is actually the most efficient solution to unbounded knapsack problem in the next section. To avoid this we have two choices 1) by using a temporary one-dimensional new dp for each i . 2) by updating the cost reversely we can make sure each time we are not using the newly updated result.

```

1 def knapsack01OptimizedDP1(c, v, C):
2     dp = [0 for _ in range(C+1)]
3     for i in range(len(c)):
4         new_dp = [0 for _ in range(C+1)]
5         for w in range(c[i], C+1):
6             new_dp[w] = max(dp[w], dp[w-c[i]]+v[i])
7         dp = new_dp
8     return dp[-1]
9
10 def knapsack01OptimizedDP2(c, v, C):
11     dp = [0 for _ in range(C+1)]
12     for i in range(len(c)):
13         for w in range(C, c[i]-1, -1):
14             dp[w] = max(dp[w], dp[w-c[i]]+v[i])
15     return dp[-1]

```

For the convenience of the later sections, we modularize the final code as:

```

1 def knapsack01(cost, val, C, dp):
2     for j in range(C, cost-1, -1):
3         dp[j] = max(dp[j], dp[j-cost]+val)
4     return dp
5 def knapsack01Final(c, v, C):
6     n = len(c)
7     dp = [0 for _ in range(C+1)]
8     for i in range(n):

```

```

9         knapsack01(c[i], v[i], C, dp)
10    return dp[-1]

```

29.6.2 Unbounded Knapsack

Unbounded knapsack problems where one item can be used for unlimited times only if the total cost is limited. So each item can be used at most $C/c[i]$ times.

Solution 1: Combination with DFS. Here, because one item can be used only if the cost is within restriction of the knapsack's capacity, thus when we recursively call DFS function, we do not increase the index i like we did in the 0-1 knapsack problem.

```

1 def knapsackUnboundDFS(c, v, C):
2     def combinationUnbound(s, cur_c, cur_v, ans):
3         ans[0] = max(ans[0], cur_v)
4         if s == n: return
5         for i in range(s, n):
6             if cur_c + c[i] <= C: # restriction
7                 combinationUnbound(i, cur_c + c[i], cur_v + v[i]
8 ], ans)
9     ans = [0]
10    n = len(c)
11    combinationUnbound(0, 0, 0, ans)
12    return ans[0]
13 print(knapsackUnboundDFS(c, v, C))
14 # output
15 # 10

```

Solution 2: Use 0-1 knapsack's dynamic programming. We can simply copy each item up to $C/c[i]$ times. Or we can do it better, because any positive integer can be composed by using $1, 2, 4, \dots, 2^k$. For instance, $3=1+2, 5=1+4, 6=2+4$. Thus we can shrink the $C/c[i]$ to $\log_2(C/c[i]) + 1$ items, each with value $c[i], v[i]; 2*c[i], 2*v[i]$, to 2^k times the cost and value.

```

1 import math
2 def knapsackUnboundNaiveDP2(c, v, C):
3     n = len(c)
4     dp = [0 for _ in range(C+1)]
5     for i in range(n):
6         for j in range(int(math.log(C/c[i], 2))+1): # call it
7             multiple times
8             # log(3, 2) = 1.4, 3 = 1+2, so we need 2, 4 = 4.
9             knapsack01(c[i]<<j, v[i]<<j, C, dp)
10    return dp[-1]
11 # output
12 # 10

```

Solution 3: Use the covered updating of the one-dimensional dp. As we mentioned in the above section, if we use one-dimensional dp without do any change of the knapsack01DP code. (still hard to explain)

```

1 def knapsackUnbound(cost , val , C, dp):
2     for j in range(cost , C+1):
3         dp[j] = max(dp[j] , dp[j-cost]+val)
4     return dp
5
6 def knapsackUnboundFinal(c , v , C):
7     n = len(c)
8     dp = [0 for _ in range(C+1)]
9     for i in range(n):
10        knapsackUnbound(c[i] , v[i] , C, dp)
11    return dp[-1]

```

29.6.3 Bounded Knapsack

In this type of problems, each item can be used at most $n[i]$ times.

Reduce to 0-1 Knapsack problem. Like in the Unbounded Knapsack, it can be reduced to 0-1 knapsack and each can appear at most $n[i]$ times. Thus, we can use $\min(\log_2(n[i]), \log_2(C/c[i]))$.

```

1 def knapsackboundDP(c , v , Num, C):
2     n = len(c)
3     dp = [0 for _ in range(C+1)]
4     for i in range(n):
5         for j in range(min(int(math.log(C/c[i] , 2))+1, int(math.
6             log(Num[i] , 2))+1)): # call it multiple times
7             knapsack01(c[i]<<j , v[i]<<j , C, dp)
8     return dp[-1]
9 Num = [2, 3, 2, 2]
10 print(knapsackboundDP(c , v , Num, C))
11 # 10

```

Reduce to Unbounded Knapsack. If $n[i] \geq C/c[i], \forall i$, then the Bounded Knapsack can be reduced to Unbounded Knapsack.

29.6.4 Generalization

The four elements of the backpack problems include:

1. State: $dp[i][c]$ denotes the optimized value (maximum value, minimum items, total number) with subproblem (0,i) with cost c.
2. State transfer Function: $dp[i][c] = f(dp[i-1][c-c[i]], dp[i-1][c])$. For example, if we want:
 - maximum/min value: $f = \max/\min, dp[i-1][c-c[i]] \rightarrow dp[i-1][c-c[i]] + v[i];$
 - total possible solutions: $dp[i][c] += dp[i-1][c-c[i]]$
 - the maximum cost (how full we can fill the snapshots): $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-c[i]] + c[i])$

3. Initialize: $f[i][0] = \text{True}$; $f[0][1, \dots, \text{size}] = \text{False}$, which is explained that if we have i items, we choose 0, so we can always get size 0, if we only have 0 items, we cant fill backpack with size in range $(1, \text{size})$.
4. Answer: $\text{dp}[\text{n}-1][\text{C}-1]$.

Restriction Requires to Reach to Exact Amount of Capacity

In the above sections, we answered different type of knapsacks with the second restriction, while how about for the first restriction which requires the total cost to be exact equal to an amount S . Think about that if we are given an amount that no combination from the cost array can be added up to this amount, then it should be set to invalid, with value $\text{float}("-\text{inf}")$ for max function and $\text{float}("inf")$ for min state function in Python. For the amount of 0, the value will be valid with 0. Thus, the only difference for the first restriction lies in the initialization. Here, we give an example of Exact for the unbounded type:

```

1 def knapsackUnboundExactNaiveDP2(c, v, C):
2     n = len(c)
3     dp = [float("-inf") for _ in range(C+1)]
4     dp[0] = 0
5     for i in range(n):
6         for j in range(int(math.log(C/c[i], 2))+1): # call it
7             multiple times
8                 knapsack01(c[i]<<j, v[i]<<j, C, dp)
9     return dp[-1] if dp[-1] != float("-inf") else 0
10 c = [2, 2, 2, 7]
11 v = [1, 2, 4, 5]
12
13 C = 17
14 print(knapsackUnboundNaiveDP2(c, v, C))
15 print(knapsackUnboundExactNaiveDP2(c, v, C))
16 # output
17 # 32
18 # 25

```

29.6.5 LeetCode Problems

- 29.19 **Coin Change (L322, **)**. You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1. *Note: You may assume that you have an infinite number of each kind of coin.*

Example 1:

```

Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1

```

Example 2:

Input: coins = [2], amount = 3
 Output: -1

Solution: Unbounded Snapsack with Restriction 1 and Do Minimum Counting. First, we are required to get the fewest length of valid combination, thus the state transfer function is $dp[i][j] = \min(dp[i-1][j], dp[i-1][j-c[i]]+1)$. Second, we need to make up exact amount thus other than at cost 0, all the others are initialize with invalid value of `float("inf")` in the dp array. Third, this is an unbounded snapsack, thus we iterate the costs incrementally with one dimensional dp array to be able to use them multiple times.

```

1 def coinChange(self, coins, amount):
2     dp = [float("inf") for c in range(amount+1)]
3     dp[0] = 0
4     # unbounded sacpback problems
5     for i in range(len(coins)):
6         for a in range(coins[i], amount+1):
7             dp[a] = min(dp[a], dp[a-coins[i]]+1)
8
9     return dp[-1] if dp[-1] != float("inf") else -1

```

29.20 **Partition Equal Subset Sum (L416, **).** Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Example 1:

Input: [1, 5, 11, 5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].

Example 2:

Input: [1, 2, 3, 5]

Output: false

Explanation: The array cannot be partitioned into equal sum subsets.

Solution 1: 0-1 Snapsack. First, we compute the total sum, only if the sum is even integer that we can possibly divide it into two equal subsets. After we obtained the possible sum, we use 0-1 snap-sack where the state transfer function: $dp[i][j] = dp[i-1][j]$ or $dp[i-1][j-nums[i]]$. And the dp array is initialized as false.

```

1 def canPartition(self, nums):
2     if not nums:
3         return False

```

```

4     s = sum(nums)
5     if s%2:
6         return False
7     # 01 snapsack
8     dp = [False]*int(s/2)+1)
9     dp[0] = True
10
11    for i in range(len(nums)):
12        for j in range(int(s/2), nums[i]-1, -1):
13            dp[j] = (dp[j] or dp[j-nums[i]])
14
15    return dp[-1]

```

Solution 2: DFS with Memo. However, here we only need to check if it equals, we can do early stop with DFS.

```

1 def canPartition(self, nums):
2     if not nums:
3         return False
4     s = sum(nums)
5     if s%2:
6         return False
7     memo = {}
8     def dfs(s, t):
9         if t == 0:
10             return True
11         if t < 0:
12             return False
13         if t not in memo:
14             memo[t] = any(dfs(i+1, t-nums[i]) for i in
15                           range(s, len(nums)))
16
17     return memo[t]

```

29.7 Exercise

29.7.1 Single Sequence

Unique Binary Search Tree
 Interleaving String
 Race Car

29.7.2 Coordinate

746. Min Cost Climbing Stair (Easy)

1 On a staircase, the i -th step has some non-negative cost $\text{cost}[i]$ assigned (0 indexed).
 2

```

3 Once you pay the cost , you can either climb one or two steps .
    You need to find minimum cost to reach the top of the floor ,
    and you can either start from the step with index 0, or the
    step with index 1.

4
5 Example 1:

6
7 Input: cost = [10 , 15 , 20]
8 Output: 15
9 Explanation: Cheapest is start on cost[1] , pay that cost and go
    to the top.

10
11 Example 2:

12
13 Input: cost = [1 , 100 , 1 , 1 , 1 , 100 , 1 , 1 , 100 , 1]
14 Output: 6
15 Explanation: Cheapest is start on cost[0] , and only step on 1s ,
    skipping cost[3].
16
17 Note:
18
19     cost will have a length in the range [2 , 1000].
20     Every cost[i] will be an integer in the range [0 , 999].

```

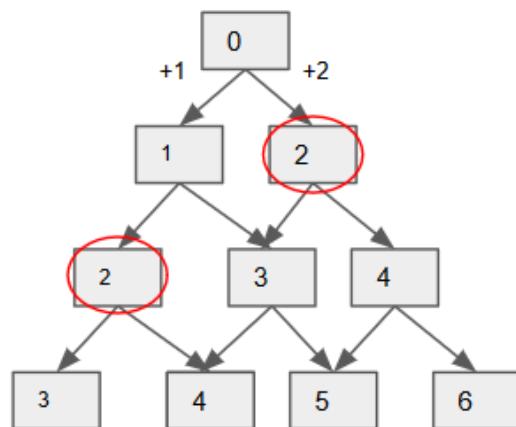


Figure 29.9: Caption

Analysis: As Fig 29.9 shows, the mincost to get 2 is only dependent on the mincost of 0, 1, each goes 2 and 1 steps respectively. To get 3, is only dependent on the mincost of 1, 2. For 0, 1, the cost is initialize as 0 because it is the starting point.

```

1 def minCostClimbingStairs(self , cost):
2     if not cost:
3         return 0
4     dp = [sys.maxsize]*(len(cost)+1)
5
6     dp[0] = 0

```

```

7 dp[1] = 0
8 for i in range(2, len(cost)+1):
9     dp[i] = min(dp[i], dp[i-1]+cost[i-1], dp[i-2]+cost[i-2])
10    return dp[-1]

```

576. Out of Boundary Paths (Medium)

1 There is an m by n grid with a ball. Given the start coordinate (i, j) of the ball, you can move the ball to adjacent cell or cross the grid boundary in four directions (up, down, left, right). However, you can at most move N times. Find out the number of paths to move the ball out of grid boundary. The answer may be very large, return it after mod $10^9 + 7$.

2

3 Example 1:

4 Input: m = 2, n = 2, N = 2, i = 0, j = 0

5 Output: 6

6 Explanation:

7

8 Example 2:

9 Input: m = 1, n = 3, N = 3, i = 0, j = 1

10 Output: 12

11 Explanation:

12

13 Note:

14 Once you move the ball out of boundary, you cannot move it back.

15 The length and height of the grid is in range [1, 50].

16 N is in range [0, 50].

Multiple Time Coordinate. The only difference compared with our examples, we track the out of boundary paths each time when the next location is not within bound.

```

1 def findPaths(self, m, n, N, i, j):
2     MOD = 10**9+7
3     dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
4     dp = [[0 for _ in range(n)] for _ in range(m)]
5     dp[i][j] = 1
6     ans = 0
7
8     for step in range(N):
9         new_dp = [[0 for _ in range(n)] for _ in range(m)]
10        for x in range(m):
11            for y in range(n):
12                if dp[x][y] == 0: #only check available location
13                    at that step
14                        continue
15                    for dx, dy in dirs:
16                        nx, ny = x+dx, y+dy
17                        if 0 <= nx < m and 0 <= ny < n:
18                            new_dp[nx][ny] += dp[x][y]
19                        else:
20                            ans += dp[x][y]
21                            ans %= MOD

```

```

21     dp = new_dp
22
23     return ans

```

63. Unique Paths II

```

1 A robot is located at the top-left corner of a m x n grid ( 
    marked 'Start' in the diagram below).
2
3 The robot can only move either down or right at any point in
    time. The robot is trying to reach the bottom-right corner of
    the grid (marked 'Finish' in the diagram below).
4
5 Now consider if some obstacles are added to the grids. How many
    unique paths would there be?
6
7 An obstacle and empty space is marked as 1 and 0 respectively in
    the grid.
8
9 Note: m and n will be at most 100.
10
11 Example 1:
12
13 Input:
14 [
15     [0 ,0 ,0] ,
16     [0 ,1 ,0] ,
17     [0 ,0 ,0]
18 ]
19 Output: 2
20 Explanation:
21 There is one obstacle in the middle of the 3x3 grid above.
22 There are two ways to reach the bottom-right corner:
23 1. Right -> Right -> Down -> Down
24 2. Down -> Down -> Right -> Right

```

Coordinate.

```

1 def uniquePathsWithObstacles(self, obstacleGrid):
2     """
3         :type obstacleGrid: List[List[int]]
4         :rtype: int
5     """
6
7     if not obstacleGrid or obstacleGrid[0][0] == 1:
8         return 0
9     m, n = len(obstacleGrid), len(obstacleGrid[0])
10    dp = [[0 for c in range(n)] for r in range(m)]
11    dp[0][0] = 1 if obstacleGrid[0][0] == 0 else 0 # starting
12    point
13
14    # init col
15    for r in range(1, m):
16        dp[r][0] = dp[r-1][0] if obstacleGrid[r][0] == 0 else 0
17
18    for c in range(1, n):
19        dp[0][c] = dp[0][c-1] if obstacleGrid[0][c] == 0 else 0
20
21    for r in range(1, m):
22        for c in range(1, n):
23            if obstacleGrid[r][c] == 0:
24                dp[r][c] = dp[r-1][c] + dp[r][c-1]
25            else:
26                dp[r][c] = 0
27
28    return dp[m-1][n-1]

```

```

17     dp[0][c] = dp[0][c-1] if obstacleGrid[0][c] == 0 else 0
18
19     for r in range(1, m):
20         for c in range(1, n):
21             dp[r][c] = dp[r-1][c] + dp[r][c-1] if obstacleGrid[r]
22             ][c] == 0 else 0
23     print(dp)
24     return dp[-1][-1]

```

29.7.3 Double Sequence

712. Minimum ASCII Delete Sum for Two Strings

```

1 Given two strings s1, s2, find the lowest ASCII sum of deleted
2   characters to make two strings equal.
3
4 Example 1:
5 Input: s1 = "sea", s2 = "eat"
6 Output: 231
7 Explanation: Deleting "s" from "sea" adds the ASCII value of "s"
8   (115) to the sum.
9 Deleting "t" from "eat" adds 116 to the sum.
10 At the end, both strings are equal, and  $115 + 116 = 231$  is the
11 minimum sum possible to achieve this.
12
13 Example 2:
14
15 Input: s1 = "delete", s2 = "leet"
16 Output: 403
17 Explanation: Deleting "dee" from "delete" to turn the string
18   into "let",
19 adds  $100[d]+101[e]+101[e]$  to the sum. Deleting "e" from "leet"
20 adds  $101[e]$  to the sum.
21 At the end, both strings are equal to "let", and the answer is
22  $100+101+101+101 = 403$ .
23 If instead we turned both strings into "lee" or "eet", we would
24 get answers of 433 or 417, which are higher.
25
26 Note:
27  $0 < s1.length, s2.length \leq 1000$ .
28 All elements of each string will have an ASCII value in [97,
29 122].

```

```

1 def minimumDeleteSum(self, s1, s2):
2     word1, word2=s1,s2
3     if not word1:
4         if not word2:
5             return 0
6         else:
7             return sum([ord(c) for c in word2])
8     if not word2:
9         return sum([ord(c) for c in word1])

```

```
10
11     rows, cols=len(word1), len(word2)
12
13     dp = [[0 for col in range(cols+1)] for row in range(rows+1)]
14     for i in range(1,rows+1):
15         dp[i][0] = dp[i-1][0] + ord(word1[i-1]) #delete in word1
16     for j in range(1,cols+1):
17         dp[0][j] = dp[0][j-1] + ord(word2[j-1]) #delete in word2
18
19     for i in range(1,rows+1):
20         for j in range(1,cols+1):
21             if word1[i-1] == word2[j-1]:
22                 dp[i][j] = dp[i-1][j-1]
23             else:
24                 dp[i][j] = min(dp[i][j-1] + ord(word2[j-1]), dp[
25 i-1][j] + ord(word1[i-1])) #delete in word2, delete in word1
      return dp[rows][cols]
```

Part VIII

Appendix

Cool Python Guide

In this chapter, instead of installing Python 3 on your device, we can use the IDE offered by greeksforgeeks website which can be found <https://ide.geeksforgeeks.org/index.php>, Google Colab is a good place to write Python notebook style document.

Python is one of the easiest advanced scripting and object-oriented programming languages to learn. Its intuitive structures and semantics are originally designed for people for are computer scientists. Python is one of the most widely used programming languages due to its:

- Easy syntax and hide concept like pointers: both non-programmers and programmers can learn to code easily and at a faster rate,
- Readability: Python is often referred as “executable pseudo-code” because its syntax mostly follows the conventions used for programmers to outline their ideas.
- Cross-platform: Python runs on all major operating systems such as Microsoft Windows, Linux, and Mac OS X
- Extensible: In addition to the standard libraries there are extensive collections of freely available add-on modules, libraries, frameworks, and tool-kits.

Compared with other programming languages, Python code is typically 3-5 times shorter than equivalent Java code, and often 5-10 times shorter than equivalent C++ code according to www.python.org. All of these simplicity and efficiency make Python an ideal language to learn under current trend, and it is also an ideal candidate language to use during Coding Interviews which is time-limited.

30.1 Python Overview

In this section, we provide a well-organized overview of how Python works as an **object-oriented programming language** in Subsection 30.1.1, what is the components of Python: built-in Data types, built-in modules, third party packages/libraries, frameworks (Subsection 30.1.2). And in this book, we selectively introduce the most useful ones that considered for the purpose of learning algorithms and passing coding interviews.

30.1.1 Understanding Objects and Operations

Basics

Everything is Object Python built-in Data Types, Modules, Classes, they are all objects. An object is a class, thus different object is just a different class. For example, we created an instance of `int` object with value 1:

```
1 >>> 1
2 1
```

We use `type()` built-in function to see its underlying type—class, for example:

```
1 >>> type([1,2,3,4])
2 <class 'list'>
3 >>> type(1)
4 <class 'int'>
5 >>> type([1,2,3,4])
6 <class 'list'>
7 >>> type(range(10))
8 <class 'range'>
9 >>> type(1)
10 <class 'int'>
11 >>> type('abc')
12 <class 'str'>
```

Operators Operators are used to perform operations on variables and instance of objects. The example shows operator `+` performed on two instances of objects.

```
1 >>> [1, 2, 3] + [4, 5, 6]
2 [1, 2, 3, 4, 5, 6]
```

Variables When we are creating an instance of objects, a common practice is to have variables which are essentially pointers pointing to the instance of object's location in memory.

```
1 >>> a = [1, 2, 3]
2 >>> b = [4, 5, 6]
3 >>> c = a + b
```

```
4 >>> c
5 [1, 2, 3, 4, 5, 6]
```

Tools To be able to help ourselves; knowing what attributes, built-in function that, we use built-in function `dir(object)`.

To check object or function's information, we use built-in function `help`. And when we are done with viewing, type `q` to exit.

Properties

In-place VS Standard Operations In-place operation is an operation that changes directly the content of a given linear algebra, vector, matrices(Tensor) without making a copy. The operators which helps to do the operation is called in-place operator. Eg: `a+= b` is equivalent to `a=operator.iadd(a, b)`. A standard operation, on the other hand, will return a new instance of object.

Mutable VS Immutable Objects All objects can be either mutable or immutable. Simply put, for immutable data types/objects, we can not add, remove, or replace its content on the fly, whereas the mutable objects can not but rather return new objects when attempting to update. Custom classes are generally mutable. The different behavior of mutable and immutable objects can be shown by using operations. A in-place operation can only be performed on mutable objects.

Object, Type, Identity, Value Everything in Python is an object including different data types, modules, classes, and functions. Each object in Python has a **type**, a **value**, and an **identity**. When we are creating an instance of an object such as string with value 'abc' it automatically comes with an "identifier". The identifier of the object acts as a pointer to the object's location in memory. The built-in function `id()` can be used to return the identity of an object as an integer which usually corresponds to the object's location in memory. `is` identity operator can be used directly to compare the identity of two objects. The built-in function `type()` can return the type of an object and operator `==` can be used to see if two objects has the same value.

Examples

Behavior of Mutable Objects Let us see an example, we create three variables/instances `a`, `b`, `c`, and `a`, `b` are assigned with object of the same value, and `c` is assigned with variable `a`.

```
1 >>> a = [1, 2, 3]
2 >>> b = [1, 2, 3]
```

```

3 >>> c = a
4 >>> id(a), id(b), id(c)
5 (140222162413704, 140222017592328, 140222162413704)

```

We use our introduced function and operators to demonstrate its behavior. First, check **a** and **b**:

```

1 >>> a == b, a is b, type(a) is type(b)
2 (True, False, True)

```

We see that **a** and **b** are having different identity, meaning the object of each points to different location in memory, they are indeed two independent objects. Now, let us compare **a** and **c** the same way:

```

1 >>> a == c, a is c, type(a) is type(c)
2 (True, True, True)

```

Ta-daa! They have the same identity, meaning they point to the same piece of memory and **c** is more like an alias to **a**. Now, let's change a value in **a** use in-place operation and see its ids:

```

1 >>> a[2] = 4
2 >>> id(a), id(b), id(c)
3 (140222162413704, 140222017592328, 140222162413704)
4 >>> a += [5]
5 >>> id(a), id(b), id(c)
6 (140222162413704, 140222017592328, 140222162413704)

```

We do not see any change about identity but change of values. Now, let us use other standard operations and see the behavior:

```

1 >>> a = a + [5]
2 >>> a
3 [1, 2, 4, 5, 5]
4 >>> id(a), id(b), id(c)
5 (140222017592392, 140222017592328, 140222162413704)

```

Now, we see **a** has a different **id** compared with **c**, meaning they are no longer the same instance of the same object any more.

Behavior of Immutable Objects For the mutable objects, we see the the reassignment of **c** to **a** results having same identity, however, this is not the case in the immutable objects, see an example:

```

1 >>> a = 'abc'
2 >>> b = 'abc'
3 >>> c = a
4 >>> id(a), id(b), id(c)
5 (140222162341424, 140222162341424, 140222162341424)

```

These three variables **a**, **b**, **c** all share the same identity, meaning they all point to the same instance of object in the same piece of memory. This ends up more efficient usage of memory. Now, let's try to change the value of the variable **a**. We called **+=** operator which is in-place operator for mutable objects:

```

1 >>> a += 'd'
2 >>> a
3 'abcd'
4 >>> id(a), id(b), id(c)
5 (140222017638952, 140222162341424, 140222162341424)

```

We see still a new instance of string object is created and with a new id 140222017638952.

30.1.2 Python Components

The plethora of built-in data types, built-in modules, third party modules or package/libraries, and frameworks contributes to the popularity and efficiency of coding in Python.

Python Data Types Python contains 12 built-in data types. These include four scalar data types(**int**, **float**, **complex** and **bool**), four sequence types(**string**, **list**, **tuple** and **range**), one mapping type(**dict**) and two set types(**set** and **frozenset**). All the four scalar data types together with string, tuple, range and frozenset are immutable, and the others are mutable. Each of these can be manipulated using:

- Operators
- Functions
- Data-type methods

Module is a file which contains python functions, global variables etc. It is nothing but .py file which has python executable code / statement. With the build-in modules, we do not need to install external packages or include these .py files explicitly in our Python project, all we need to do is importing them directly and use their objects and corresponding methods. For example, we use built-in module Array:

```

1 import Array
2 # use it

```

We can also write a .py file ourselves and import them. We provide reference to some of the popular and useful built-in modules that is not covered in Part ?? in Python in Section 30.9 of this chapter, they are:

- Re

Package/Library Package or library is namespace which contains multiple package/modules. It is a directory which contains a special file `__init__.py`

Let's create a directory user. Now this package contains multiple packages / modules to handle user related requests.

```

user/      # top level package
    __init__.py

    get/      # firstsubpackage
        __init__.py
        info.py
        points.py
        transactions.py

    create/   # secondsubpackage
        __init__.py
        api.py
        platform.py

```

Now you can import it in following way

```

1 from user.get import info # imports info module from get package
2 from user.create import api #imports api module from create
    package

```

When we import any package, python interpreter searches for sub directories / packages.

Library is collection of various packages. There is no difference between package and python library conceptually. Have a look at requests/requests library. We use it as a package.

Framework It is a collection of various libraries which architects the code flow. Let's take example of Django which has various in-built libraries like Auth, user, database connector etc. Also, in artificial intelligence field, we have TensorFlow, PyTorch, SkLearn framework to use.

When Mutability Matters

Mutability might seem like an innocuous topic, but when writing an efficient program it is essential to understand. For instance, the following code is a straightforward solution to concatenate a string together:

```

1 string_build = ""
2 for data in container:
3     string_build += str(data)

```

In reality, this is very *inefficient*. Because strings are immutable, concatenating two strings together actually creates a third string which is the combination of the previous two. If you are iterating a lot and building a large string, you will waste a lot of memory creating and throwing away objects. Also, at the end of the iteration you will be allocating and throwing away very large string objects which is even more costly.

The following is a more efficient and pythonic way:

```

1 builder_list = []
2 for data in container:

```

```

3     builder_list.append(str(data))
4     """.join(builder_list)
5
6 ##### Another way is to use a list comprehension
7     """.join([str(data) for data in container])
8
9 ##### or use the map function
10    """.join(map(str, container))

```

This code takes advantage of the mutability of a single list object to gather your data together and then allocate a single result string to put your data in. That cuts down on the total number of objects allocated by almost half.

Another pitfall related to mutability is the following scenario:

```

1 def my_function(param=[]):
2     param.append("thing")
3     return param
4
5 my_function() # returns ["thing"]
6 my_function() # returns ["thing", "thing"]

```

What you might think would happen is that by giving an empty list as a default value to param, a new empty list is allocated each time the function is called and no list is passed in. But what actually happens is that every call that uses the default list will be using the same list. This is because Python (a) only evaluates functions definitions once, (b) evaluates default arguments as part of the function definition, and (c) allocates one mutable list for every call of that function.

Do not put a mutable object as the default value of a function parameter. Immutable types are perfectly safe. If you want to get the intended effect, do this instead:

```

1 def my_function2(param=None):
2     if param is None:
3         param = []
4     param.append("thing")
5     return param
6 Conclusion

```

Mutability matters. Learn it. Primitive-like types are probably immutable. Container-like types are probably mutable.

30.2 Data Types and Operators

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand. Python offers Arithmetic operators, Assignment Operator, Comparison Operators, Logical Operators, Bitwise Operators (shown in Chapter III), and two special operators like the identity operator or the membership operator.

30.2.1 Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Table 30.1: Arithmetic operators in Python

Operator	Description	Example
+	Add two operands or unary plus	$x + y+2$
-	Subtract right operand from the left or unary minus	$x - y -2$
*	Multiply two operands	$x * y$
/	Divide left operand by the right one (always results into float)	x / y
//	Floor division - division that results into whole number adjusted to the left in the number line	$x // y$
**	Exponent - left operand raised to the power of right	$x**y$ (x to the power y)
%	Modulus - Divides left hand operand by right hand operand and returns remainder	$x \% y$

30.2.2 Assignment Operators

Assignment operators are used in Python to assign values to variables.

`a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable `a` on the left.

There are various compound operators that follows the order: `variable_name (arithmetic operator) = variable or data type`. Such as `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

30.2.3 Comparison Operators

Comparison operators are used to compare values. It either returns True or False according to the condition.

30.2.4 Logical Operators

Logical operators are the *and*, *or*, *not* operators. It is important for us to understand what are the values that Python considers False and True. The following values are considered False, and all the other values are considered *True*.

- The *None* type

Table 30.2: Comparison operators in Python

Operator	Description	Example
>	Greater than - True if left operand is greater than the right	x > y
<	Less than - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

- Boolean False
- An integer, float, or complex zero
- An empty sequence or mapping data type
- An instance of a user-defined class that defines a `__len__()` or `__bool__()` method that returns zero or False.

Table 30.3: Logical operators in Python

Operator	Description	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

30.2.5 Special Operators

Python language offers some special type of operators like the identity operator or the membership operator.

Identity operators Identity operators are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical as we have shown in the last section.

Table 30.4: Identity operators in Python

Operator	Description	Example
is	True if the operands are identical (refer to the same object)	x is y
is not	True if the operands are not identical (do not refer to the same object)	x is not y

Membership Operators *in* and *notin* are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

Table 30.5: Membership operators in Python

Operator	Description	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

30.3 Function

30.3.1 Python Built-in Functions

Check out here <https://docs.python.org/3/library/functions.html>.

Built-in Data Types We have functions like `int()`, `float()`, `str()`, `tuple()`, `list()`, `set()`, `dict()`, `bool()`, `chr()`, `ord()`. These functions can be used for initialization, and also used for type conversion between different data types.

30.3.2 Lambda Function

In Python, *lambda function* is anonymous function, which is a function that is defined without a name. While normal functions are defined using the `def` keyword, in anonymous functions are defined using the `lambda` keyword, this is where the name comes from.

Syntax The syntax of lambda function in Python is:

```
1 lambda arguments: expression
```

Lambda function can has zero to multiple arguments but only one expression, which will be evaluated and returned. For example, we define a lambda function which takes one argument x and return x^2 .

```
1 square1 = lambda x: x**2
```

The above lambda function is equal to a normal function defined as:

```
1 def square(x):
2     return x**2
```

Calling the following code has the same output:

```
1 square1(5) == square(5)
```

Applications Hence, anonymous functions are also called lambda functions. The use of lambda creates an anonymous function (which is callable). In the case of sorted the callable only takes one parameters. Python's lambda is pretty simple. It can only do and return one thing really.

The syntax of lambda is the word lambda followed by the list of parameter names then a single block of code. The parameter list and code block are delineated by colon. This is similar to other constructs in python as well such as while, for, if and so on. They are all statements that typically have a code block. Lambda is just another instance of a statement with a code block.

We can compare the use of lambda with that of def to create a function.

```
1 adder_lambda = lambda parameter1, parameter2: parameter1+
    parameter2
```

The above code equals to the following:

```
1 def adder_regular(parameter1, parameter2):
2     return parameter1+parameter2
```

30.3.3 Map, Filter and Reduce

These are three functions which facilitate a functional approach to programming. We will discuss them one by one and understand their use cases.

Map

Map applies a function to all the items in an input_list. Here is the blueprint:

```
1 map(function_to_apply, list_of_inputs)
```

Most of the times we want to pass all the list elements to a function one-by-one and then collect the output. For instance:

```
1 items = [1, 2, 3, 4, 5]
2 squared = []
3 for i in items:
4     squared.append(i**2)
```

Map allows us to implement this in a much simpler and nicer way. Here you go:

```

1 items = [1, 2, 3, 4, 5]
2 squared = list(map(lambda x: x**2, items))

```

Most of the times we use lambdas with map so I did the same. Instead of a list of inputs we can even have a list of functions! Here we use $x(i)$ to call the function, where x is replaced with each function in funcs, and i is the input to the function.

```

1 def multiply(x):
2     return (x*x)
3 def add(x):
4     return (x+x)
5
6 funcs = [multiply, add]
7 for i in range(5):
8     value = list(map(lambda x: x(i), funcs))
9     print(value)
10
11 # Output:
12 # [0, 0]
13 # [1, 2]
14 # [4, 4]
15 # [9, 6]
16 # [16, 8]

```

Filter

As the name suggests, filter creates a list of elements for which a function returns true. Here is a short and concise example:

```

1 number_list = range(-5, 5)
2 less_than_zero = list(filter(lambda x: x < 0, number_list))
3 print(less_than_zero)
4
5 # Output: [-5, -4, -3, -2, -1]

```

The filter resembles a for loop but it is a builtin function and faster.

Note: If map and filter do not appear beautiful to you then you can read about list/dict/tuple comprehensions.

Reduce

Reduce is a really useful function for performing some computation on a list and returning the result. It applies a rolling computation to sequential pairs of values in a list. For example, if you wanted to compute the product of a list of integers.

So the normal way you might go about doing this task in python is using a basic for loop:

```

1 product = 1
2 list = [1, 2, 3, 4]
3 for num in list:

```

```

4     product = product * num
5
6 # product = 24

```

Now let's try it with reduce:

```

1 from functools import reduce
2 product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
3
4 # Output: 24

```

30.4 Class

30.4.1 Special Methods

From [1]. <http://www.informit.com/articles/article.aspx?p=453682&seqNum=6> All the built-in data types implement a collection of special object methods. The names of special methods are always preceded and followed by double underscores (`__`). These methods are automatically triggered by the interpreter as a program executes. For example, the operation $x + y$ is mapped to an internal method, `x.__add__(y)`, and an indexing operation, `x[k]`, is mapped to `x.__getitem__(k)`. The behavior of each data type depends entirely on the set of special methods that it implements.

User-defined classes can define new objects that behave like the built-in types simply by supplying an appropriate subset of the special methods described in this section. In addition, built-in types such as lists and dictionaries can be specialized (via inheritance) by redefining some of the special methods. In this book, we only list the essential ones so that it speeds up our interview preparation.

Object Creation, Destruction, and Representation We first list these special methods in Table 30.6. A good and useful way to implement a class is through `__repr__()` method. By calling built-in function `repr(built-in object)` and implement self-defined class as the same as built-in object. Doing so avoids us implementing a lot of other special methods for our class and still has most of behaviors needed. For example, we define a Student class and represent it as of a tuple:

```

1 class Student:
2     def __init__(self, name, grade, age):
3         self.name = name
4         self.grade = grade
5         self.age = age
6     def __repr__(self):
7         return repr((self.name, self.grade, self.age))
8 a = Student('John', 'A', 14)
9 print(hash(a))
10 print(a)

```

Table 30.6: Special Methods for Object Creation, Destruction, and Representation

Method	Description
<code>*__init__(self, *args, **kwargs)</code>	Called to initialize a new instance
<code>__del__(self)</code>	Called to destroy an instance
<code>*__repr__(self)</code>	Creates a full string representation of an object
<code>__str__(self)</code>	Creates an informal string representation
<code>__cmp__(self, other)</code>	Compares two objects and returns negative, zero, or positive
<code>__hash__(self)</code>	Computes a 32-bit hash index
<code>hline</code>	Returns 0 or 1 for truth-value testing
<code>__nonzero__(self)</code>	
<code>__unicode__(self)</code>	Creates a Unicode string representation

If we have no `__repr__()`, the output for the following test cases are:

```
1 8766662474223
2 <__main__.Student object at 0x7f925cd79ef0>
```

Doing so, we has `__hash__()`,

Comparison Operations Table 30.7 lists all the comparison methods that might need to be implemented in a class in order to apply comparison in applications such as sorting.

Table 30.7: Special Methods for Object Creation, Destruction, and Representation

Method	Description
<code>__lt__(self, other)</code>	<code>self < other</code>
<code>__le__(self, other)</code>	<code>self <= other</code>
<code>__gt__(self, other)</code>	<code>self > other</code>
<code>__ge__(self, other)</code>	<code>self >= other</code>
<code>__eq__(self, other)</code>	<code>self == other</code>
<code>__ne__(self, other)</code>	<code>self != other</code>

30.4.2 Class Syntax

30.4.3 Nested Class

When we solving problem on leetcode, sometimes we need to wrap another class object inside of the solution class. We can do this with the nested class. When we're newing an instance, we use `mainClassName.NestedClassName()`.

30.5 Shallow Copy and the deep copy

For list and string data structures, we constantly met the case that we need to copy. However, in programming language like C++, Python, we need to know the difference between shallow copy and deep copy. Here we only introduce the Python version.

Given the following two snippets of Python code:

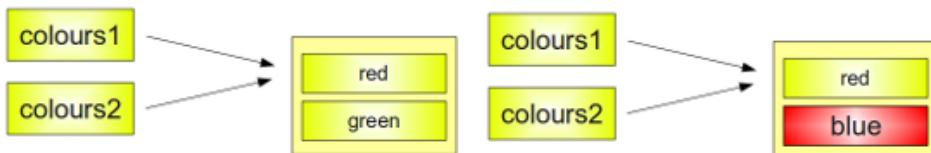
```

1 colours1 = [ "red" , "green" ]
2 colours2 = colours1
3 colours2 = [ "rouge" , "vert" ]
4 print(colours1)
5 >>> [ 'red' , 'green' ]
```

```

1 colours1 = [ "red" , "green" ]
2 colours2 = colours1
3 colours2[1] = "blue"
4 print(colours1)
5 [ 'red' , 'blue' ]
```

From the above outputs, we can see that the colors1 list is the same but in the second case, it is changed although we are assigning value to colors2. The result can be either wanted or not wanted. In python, to assign one list to other directly is similar to a pointer in C++, which both point to the same physical address. In the first case, colors2 is reassigned a new list, which has a new address, so now colors2 points to the address of this new list instead, which leaves the values of colors2 untouched at all. We can visualize this process as follows: However, we often need to do copy and



(a) The copy process for code 1

(b) The copy process for code 2

Figure 30.1: Copy process

leave the original list or string unchanged. Because there are a variety of list, from one dimensional, two-dimensional to multi-dimensional.

30.5.1 Shallow Copy using Slice Operator

It's possible to completely copy shallow list structures with the slice operator without having any of the side effects, which we have described above:

```

1 list1 = [ 'a' , 'b' , 'c' , 'd' ]
2 list2 = list1 [:]
3 list2[1] = 'x'
```

```

4 print(list2)
5 ['a', 'x', 'c', 'd']
6 print(list1)
7 ['a', 'b', 'c', 'd']

```

Also, for Python 3, we can use `list.copy()` method

```
1 list2 = list1.copy()
```

But as soon as a list contains sublists, we have the same difficulty, i.e. just pointers to the sublists.

```

1 lst1 = ['a', 'b', ['ab', 'ba']]
2 lst2 = lst1 [:]

```

This behaviour is depicted in the following diagram:

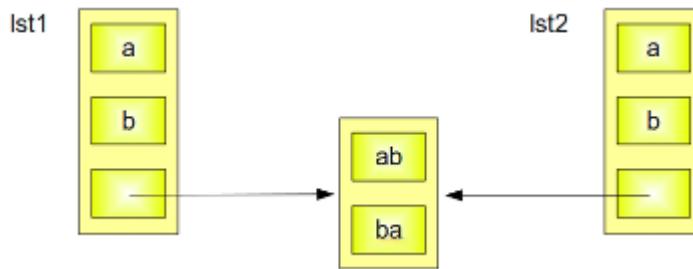


Figure 30.2: Caption

If you assign a new value to the 0th Element of one of the two lists, there will be no side effect. Problems arise, if you change one of the elements of the sublist.

```

1 >>> lst1 = ['a', 'b', ['ab', 'ba']]
2 >>> lst2 = lst1 [:]
3 >>> lst2[0] = 'c'
4 >>> lst2[2][1] = 'd'
5 >>> print(lst1)
6 ['a', 'b', ['ab', 'd']]

```

The following diagram depicts what happens, if one of the elements of a sublist will be changed: Both the content of `lst1` and `lst2` are changed.

30.5.2 Iterables, Generators, and Yield

<https://pythontips.com/2013/09/29/the-python-yield-keyword-explained/>. Seems like it can not yeild a list.

30.5.3 Deep Copy using copy Module

A solution to the described problems is to use the module "copy". This module provides the method "copy", which allows a complete copy of a arbitrary list, i.e. shallow and other lists.

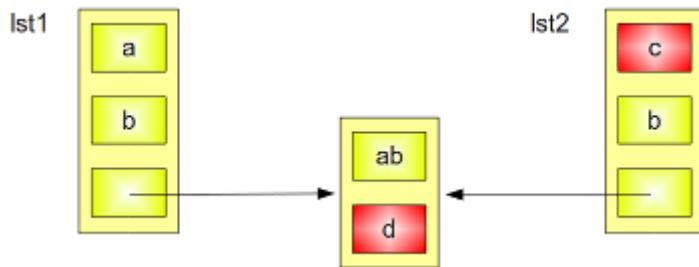


Figure 30.3: Caption

The following script uses our example above and this method:

```

1 from copy import deepcopy
2
3 lst1 = [ 'a' , 'b' , [ 'ab' , 'ba' ] ]
4
5 lst2 = deepcopy(lst1)
6
7 lst2 [ 2 ][ 1 ] = "d"
8 lst2 [ 0 ] = "c";
9
10 print lst2
11 print lst1

```

If we save this script under the name of deep_copy.py and if we call the script with “python deep_copy.p”, we will receive the following output:

```

1 $ python deep_copy.py
2 [ 'c' , 'b' , [ 'ab' , 'd' ] ]
3 [ 'a' , 'b' , [ 'ab' , 'ba' ] ]

```

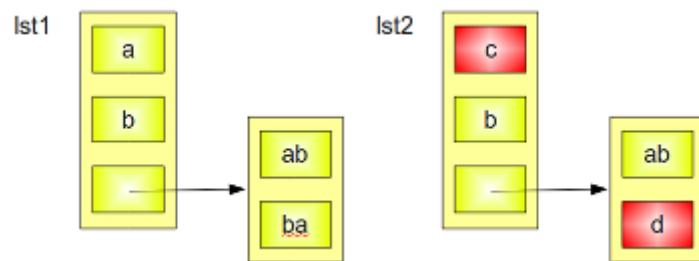


Figure 30.4: Caption

This section is cited from Need to modify.

30.6 Global Vs nonlocal

30.7 Loops

The for loop can often be needed in algorithms we have two choices: *for* and *while*. So to learn the basic grammar to do for loop easily could help us be more efficient in programming.

Usually for loop is used to iterate over a sequence or matrix data. For example, the following grammar works for either string or list.

```

1 # for loop in a list to get the value directly
2 a = [5, 4, 3, 2, 1]
3 for num in a:
4     print(num)
5 # for loop in a list use index
6 for idx in range(len(a)):
7     print(a[idx])
8 # for loop in a list get both index and value directly
9 for idx, num in enumerate(a):
10    print(idx, num)
```

Sometimes, we want to iterate two lists jointly at the same time, which requires they both have the same length. We can use *zip* to join them together, and all the others for loop works just as the above. For example:

```

1 a, b = [1, 2, 3, 4, 5], [5, 4, 3, 2, 1]
2 for idx, (num_a, num_b) in enumerate(zip(a, b)):
3     print(idx, num_a, num_b)
```

30.8 Special Skills

1. Swap the value of variable

```

1 a, b = 7, 10
2 print(a, b)
3 a, b = b, a
4 print(a, b)
```

2. Join all the string elements in a list to a whole string

```

1 a = ["Cracking", "LeetCode", "Problems"]
2 print("".join(a))
3
```

3. Find the most frequent element in a list

```

1 a = [1, 3, 5, 6, 9, 9, 4, 10, 9]
2 print(max(set(a), key=a.count))
3 # or use counter from the collections
4 from collections import Counter
```

```

5     cnt = Counter(a)
6     print(cnt.most_common(3))
7

```

4. Check if two strings are comprised of the same letters.

```

1     from collections import Counter
2     Counter(str1) == Counter(str2)
3

```

5. Reversing

```

1     # 1. reversing strings or list
2     a = 'crackingleetcode'
3     b = [1, 2, 3, 4, 5]
4     print(a[::-1], a[::-1])
5     # 2. iterate over each char of the string or list
6     # contents in reverse order efficiently, here we use zip
7     # to
8     for char, num in zip(reversed(a), reversed(b)):
9         print(char, num)
10    #3. reverse each digit in an integer or float number
11    num = 123456789
12    print(int(str(num)[::-1]))
13

```

6. Remove the duplicates from list or string. We can convert it to set at first, but this wont keep the original order of the elements. If we want to keep the order, we can use the OrderdDict method from collections.

```

1     a = [5, 4, 4, 3, 3, 2, 1]
2     no_duplicate = list(set(a))
3
4     from collections import OrderedDict
5     print(list(OrderedDict.fromkeys(a).keys()))
6

```

7. Find the min or max element or the index.

30.9 Supplemental Python Tools

30.9.1 Re

30.9.2 Bitsect

```

1 def index(a, x):
2     'Locate the leftmost value exactly equal to x'
3     i = bisect_left(a, x)
4     if i != len(a) and a[i] == x:
5         return i
6     raise ValueError

```

```

7
8 def find_lt(a, x):
9     'Find rightmost value less than x'
10    i = bisect_left(a, x)
11    if i:
12        return a[i-1]
13    raise ValueError
14
15 def find_le(a, x):
16     'Find rightmost value less than or equal to x'
17    i = bisect_right(a, x)
18    if i:
19        return a[i-1]
20    raise ValueError
21
22 def find_gt(a, x):
23     'Find leftmost value greater than x'
24    i = bisect_right(a, x)
25    if i != len(a):
26        return a[i]
27    raise ValueError
28
29 def find_ge(a, x):
30     'Find leftmost item greater than or equal to x'
31    i = bisect_left(a, x)
32    if i != len(a):
33        return a[i]
34    raise ValueError

```

30.9.3 collections

collections is a module in Python that implements specialized container data types alternative to Python's general purpose built-in containers: dict, list, set, and tuple. The including container type is summarized in Table 30.8. Most of them we have learned in Part ??, therefore, in the table we simply put the reference in the table. Before we use them, we need to import each data type as:

```

1 from collections import deque, Counter, OrderedDict, defaultdict
, namedtuple

```

Table 30.8: Container Data types in **collections** module.

Container	Description	Refer
namedtuple	factory function for creating tuple subclasses with named fields	
deque	list-like container with fast appends and pops on either end	
Counter	dict subclass for counting hashable objects	
defaultdict	dict subclass that calls a factory function to supply missing values	
OrderedDict	dict subclass that remembers the order entries were added	

Bibliography

- [1] D. M. Beazley, *Python essential reference*, Addison-Wesley Professional, 2009.
- [2] T. H. Cormen, *Introduction to algorithms*, MIT press, 2009.
- [3] S. Halim and F. Halim, *Competitive Programming 3*, Lulu Independent Publish, 2013.
- [4] B. Slatkin, *Effective Python: 59 Specific Ways to Write Better Python*, Pearson Education, 2015.
- [5] H. hua jiang, “Leetcode blogs,” <https://zxi.mytechroad.com/blog/category>, 2018, [Online; accessed 19-July-2018].
- [6] B. Baka, “Python data structures and algorithms: Improve application performance with graphs, stacks, and queues,” 2017.
- [7] “Competitive Programming,” <https://cp-algorithms.com/>, 2019, [Online; accessed 19-July-2018].
- [8] “cs princeton,” <https://aofa.cs.princeton.edu/60trees/>, 2019, [Online; accessed 19-July-2018].
- [9] S. S. Skiena, *The algorithm design manual: Text*, vol. 1, Springer Science & Business Media, 1998.
- [10] D. Phillips, *Python 3 Object Oriented Programming*, Packt Publishing Ltd, 2010.