

Preparing for the Real-world Software Engineering

Mastering Data Structures, Algorithms,
Problem-patterns and Python along
Cracking the Coding Interview

LI YIN¹

July 29, 2019

¹<https://liyinscience.com>

Dedicated to my parents, Qiongzhen Xie and
Xueping Yin, and my lovely dog, Apple.

Contents

0 Preface	1
0.0.1 First Stage	7
0.0.2 Second Stage	7
0.0.3 Third Stage	7
0.0.4 Fourth Stage	7
I Introduction	9
1 Coding Interview and Resources	11
1.1 Coding Interviews	11
1.1.1 Process of Interviews	11
1.1.2 The Role of Coding Interviews	11
1.2 Tips and Resources	11
1.2.1 Tips to Preparation	12
1.2.2 Resources	12
1.2.3 5 Tips to the Process	13
1.3 LeetCode	14
1.3.1 Special Features	14
2 The Global Picture	19
2.1 Problem-formulation	21
2.2 Searching Algorithms	23
2.3 Algorithm Design Paradigms	24
2.4 Problem Modeling	26

II Fundamental Algorithm Design and Analysis	29
3 Iteration and Recursion	33
3.1 Iteration VS Recursion	33
3.2 Recursion	34
3.3 Summary	35
4 Divide and Conquer	37
4.1 Dividing and Recurrence Function	38
4.2 Divide and Conquer	41
4.3 More Examples	42
5 Complexity Analysis	45
5.1 Notations and Definitions	46
5.2 Three Cases Time Complexity Analysis	51
5.3 Solve Recurrence Function	52
5.3.1 Iteration and Recursion Tree	54
5.3.2 Characteristic Root Technique	55
5.3.3 Master Method	57
5.4 Amortized Analysis	58
5.5 Space Complexity	58
5.5.1 Summary	58
5.5.2 More Examples	58
5.6 Time VS Space Complexity	60
5.6.1 Big-O Cheat Sheet	60
5.7 Exercises	61
5.7.1 Knowledge Check	61
III Footstone: Bit Manipulation and Data Structures	63
6 Bit Manipulation	67
6.1 Python Bitwise Operators	67
6.2 Python Built-in Functions	69
6.3 Twos-complement Binary	70
6.4 Useful Combined Bit Operations	72
6.5 Applications	74
6.6 Exercises	79
7 Linear Data Structure	81
7.1 Array	81
7.1.1 Python Built-in Sequence: List, Tuple, String, and Range	83
7.1.2 Bonus	90
7.1.3 Exercises	90

7.2	Linked List	92
7.2.1	Singly Linked List	92
7.2.2	Doubly Linked List	95
7.2.3	Bonus	96
7.2.4	Exercises	98
7.3	Stack and Queue	99
7.3.1	Basic Implementation	100
7.3.2	Deque: Double-Ended Queue	102
7.3.3	Python built-in Module: Queue	104
7.3.4	Monotone Stack	105
7.3.5	Bonus	110
7.3.6	Exercises	111
7.4	Hash Table	112
7.4.1	Hash Function Design	114
7.4.2	Collision Resolution	115
7.4.3	Implementation	117
7.4.4	Python Built-in Data Structures	120
7.4.5	Exercises	123
8	Trees	125
8.1	Introduction and Terminologies	127
8.2	N-ary Tree and Binary Tree	129
8.3	LeetCode Problems	132
9	Graphs	133
9.1	Introduction and Terminologies	134
9.2	Graph Representation	136
9.2.1	Python Two-dimensional Array	136
9.2.2	Four Types of Graph Representation	139
9.3	Exercises	141
9.3.1	Knowledge Check	141
9.3.2	Coding Practice	142
10	Heap and Priority Queue	143
10.1	Heap	143
10.1.1	Basic Implementation	144
10.1.2	Python Built-in Library: heapq	148
10.2	Priority Queue	150
10.3	Bonus	155
10.4	Exercises	155

IV Searching Algorithms	157
11 Basic Searching	163
11.1 General Searching Strategies	163
11.1.1 Breath-first Search	164
11.1.2 Depth-first Search	166
11.1.3 Priority-first Search	168
11.2 Linear Search	168
11.3 Tree Traversal	169
11.3.1 Depth-First Tree Traversal	169
11.3.2 Breath-first Tree Traversal	173
11.4 Searching on Graph	173
11.4.1 Breadth-first Search (BFS)	174
11.4.2 Depth-First-Search (DFS)	179
11.4.3 Comparison of BFS and DFS	186
11.5 Discussion of Graph Search	186
12 Advanced Linear Data Structures-based Search	187
12.1 Binary Search	187
12.1.1 Standard Binary Search and Python Module bisect . .	188
12.1.2 Binary Search in Rotated Sorted Array	190
12.1.3 Binary Search on Result Space	192
12.1.4 LeetCode Problems	194
12.2 Two-pointer Search	197
12.2.1 Slow-fast Pointer	198
12.2.2 Opposite-directional Two pointer	201
12.2.3 Sliding Window Algorithm	202
12.2.4 LeetCode Problems	210
13 Non-linear Recursive Backtracking	211
13.1 Enumeration	214
13.1.1 Permutation	214
13.1.2 Combination	217
13.1.3 All Paths	218
13.2 Solve CSP with Search Prunning	220
13.2.1 Sudoku	220
13.2.2 Eight Queen	225
13.2.3 Travels Salesman Problems (TSP)	231
13.3 Summary	231
13.4 Bonus	232
13.5 Exercises	233
13.5.1 Knowledge Check	233
13.5.2 Coding Practice	233

14 Non-Recursive Graph Search	235
14.1 Bidirectional BFS	235
14.2	235
15 Tree-based Search	237
15.1 Binary Search Tree	238
15.2 Segment Tree	244
15.3 Trie for String	249
15.4 Bonus	255
15.5 LeetCode Problems	256
V Heuristic Search	259
16 Heuristic Search	261
16.1 Hill Climbing	261
16.2 Simulated Annealing	261
16.3 Best-first Search	261
16.4 The A^* Algorithm	261
16.5 Graceful decay of admissibility	261
16.6 Summary and comparison with Complete Search	261
VI Advanced Algorithm Design	265
17 Dynamic Programming	269
17.1 From Divide-Conquer to Dynamic Programming	271
17.1.1 Fibonacci Sequence	272
17.1.2 Longest Increasing Subsequence	273
17.2 Dynamic Programming Knowledge Base	277
17.2.1 Two properties	277
17.2.2 Four Elements	278
17.2.3 Dos and Do nots	278
17.2.4 Generalization: Steps to Solve Dynamic Programming	279
17.3 Problems Can be Optimized using DP	280
17.3.1 Example of optimizing Exponential problem	280
17.3.2 Example of optimizing Polynomial problem	283
17.3.3 Single-Choice and Multiple-Choice State	288
17.4 Time Complexity Analysis	290
17.5 Exercises	292
17.5.1 Knowledge Check	292
17.5.2 Coding Practice	292
17.6 Summary	302

18 Greedy Algorithms	305
18.1 From Dynamic Programming to Greedy Algorithm	305
18.2 Hacking Greedy Algorithm	306
18.2.1 Greedy-choice Property	307
18.2.2 Prove the Correctness of Greedy Algorithms	307
VII Advanced and Special Topics	309
19 Advanced Graph Searching Algorithms	313
19.1 Cycle Detection	313
19.2 Topological Sort in Directed Acyclic Graph	316
19.3 Connected Components	317
19.3.1 Finding Connected Components	318
19.3.2 Finding Strongly Connected Components	319
19.4 Minimum Spanning Trees	322
19.4.1 Prim Algorithm	323
19.4.2 Kruskal's Algorithm	328
19.4.3 Compare Kruskal's to Prim's Algorithm	330
19.5 Single-Source Shortest Paths	331
19.5.1 The Bellman-Ford Algorithm in General	335
19.5.2 Dijkstra's Algorithm in Non-negative Weighted DG .	337
19.5.3 General Algorithm in DAG	343
19.6 All-Pairs Shortest Paths	343
19.6.1 Bellman-Ford Extension: Matrix Multiplication like Shortest Path Algorithm	345
19.6.2 The Floyd-Warshall Algorithm	347
20 Advanced Data Structures	349
20.1 Disjoint Set	349
20.1.1 Basic Implementation with Linked-list or List	351
20.1.2 Implementation with Disjoint-set Forests	352
20.2 Fibonacci Heap	357
20.3 Exercises	357
20.3.1 Knowledge Check	357
20.3.2 Coding Practice	357
21 Dynamic Programming Special (15%)	359
21.1 Single Sequence $O(n)$	360
21.1.1 Easy Type	361
21.1.2 Subarray Sum: Prefix Sum and Kadane's Algorithm .	365
21.1.3 Subarray or Substring	369
21.1.4 Exercise	371
21.2 Single Sequence $O(n^2)$	371

21.2.1 Subsequence	371
21.2.2 Splitting	372
21.3 Single Sequence $O(n^3)$	378
21.3.1 Interval	378
21.4 Coordinate: BFS and DP	384
21.4.1 One Time Traversal	384
21.4.2 Multiple-time Traversal	390
21.4.3 Generalization	396
21.5 Double Sequence: Pattern Matching DP	397
21.5.1 Longest Common Subsequence	398
21.5.2 Other Problems	399
21.5.3 Summary	405
21.6 Knapsack	405
21.6.1 0-1 Knapsack	406
21.6.2 Unbounded Knapsack	408
21.6.3 Bounded Knapsack	409
21.6.4 Generalization	409
21.6.5 LeetCode Problems	410
21.7 Exercise	412
21.7.1 Single Sequence	412
21.7.2 Coordinate	412
21.7.3 Double Sequence	416
22 String pattern Matching Special	419
22.1 Exact Single-Pattern Matching	419
22.1.1 Prefix Function and Knuth Morris Pratt (KMP)	421
22.1.2 More Applications of Prefix Functions	427
22.1.3 Z-function	427
22.2 Exact Multi-Patterns Matching	430
22.2.1 Suffix Trie/Tree/Array Introduction	430
22.2.2 Suffix Array and Pattern Matching	430
22.2.3 Rabin-Karp Algorithm (Exact or anagram Pattern Matching)	436
22.3 Bonus	436
VIII Combinatorial Problems	439
23 Sorting and Selection Algorithms	441
23.1 $O(n^2)$ Sorting	443
23.1.1 Insertion Sort	443
23.1.2 Bubble Sort and Selection Sort	445
23.2 $O(n \log n)$ Sorting	448
23.2.1 Merge Sort	448

23.2.2 HeapSort	451
23.2.3 Quick Sort and Quick Select	451
23.3 $O(n + k)$ Counting Sort	454
23.4 $O(n)$ Sorting	456
23.4.1 Bucket Sort	456
23.4.2 Radix Sort	457
23.5 Lexicographical Order	459
23.6 Python Built-in Sort	461
23.6.1 Basic and Comparison	461
23.6.2 Customize Comparison Through Key	462
23.7 Summary and Bonus	464
23.8 LeetCode Problems	464
24 Permutation, Combination, and Partition	469
24.1 Partitions	469
24.1.1 Integer Partition	470
24.1.2 Set Partition	470
24.1.3 Traveling Salesman Problem	470
IX Math and Geometry	471
25 Math and Probability Problems	473
25.1 Numbers	473
25.1.1 Prime Numbers	473
25.1.2 Ugly Numbers	475
25.1.3 Combinatorics	477
25.2 Intersection of Numbers	480
25.2.1 Greatest Common Divisor	480
25.2.2 Lowest Common Multiple	481
25.3 Arithmetic Operations	482
25.4 Probability Theory	483
25.5 Linear Algebra	484
25.6 Geometry	484
25.7 Miscellaneous Categories	486
25.7.1 Floyd's Cycle-Finding Algorithm	486
25.8 Exercise	487
25.8.1 Number	487
X Questions by Data Structures	489
26 Array Questions(15%)	491
26.1 Subarray	492

26.1.1	Absolute-conditioned Subarray	494
26.1.2	Vague-conditioned subarray	501
26.1.3	LeetCode Problems and Misc	506
26.2	Subsequence (Medium or Hard)	510
26.2.1	Others	512
26.3	Subset(Combination and Permutation)	518
26.3.1	Combination	518
26.3.2	Combination Sum	522
26.3.3	K Sum	525
26.3.4	Permutation	530
26.4	Merge and Partition	531
26.4.1	Merge Lists	531
26.4.2	Partition Lists	531
26.5	Intervals	531
26.5.1	Speedup with Sweep Line	533
26.5.2	LeetCode Problems	536
26.6	Intersection	536
26.7	Miscellaneous Questions	537
26.8	Exercises	538
26.8.1	Subsequence with (DP)	538
26.8.2	Subset	543
26.8.3	Intersection	544
27	Linked List, Stack, Queue, and Heap Questions (12%)	545
27.1	Linked List	545
27.2	Queue and Stack	547
27.2.1	Implementing Queue and Stack	547
27.2.2	Solving Problems Using Queue	548
27.2.3	Solving Problems with Stack and Monotone Stack	549
27.3	Heap and Priority Queue	557
28	String Questions (15%)	561
28.1	Ad Hoc Single String Problems	562
28.2	String Expression	562
28.3	Advanced Single String	562
28.3.1	Palindrome	562
28.3.2	Calculator	568
28.3.3	Others	572
28.4	Exact Matching: Sliding Window and KMP	572
28.5	Anagram Matching: Sliding Window	572
28.6	Exact Matching	573
28.6.1	Longest Common Subsequence	573
28.7	Exercise	573
28.7.1	Palindrome	573

29 Tree Questions(10%)	575
29.1 Core Principle	575
29.2 N-ary Tree (40%)	577
29.2.1 Tree Traversal	577
29.2.2 Depth/Height/Diameter	581
29.2.3 Paths	584
29.2.4 Reconstruct the Tree	591
29.2.5 Find element	596
29.2.6 Ad Hoc Problems	597
29.3 Binary Search Tree (BST)	602
29.3.1 BST Rules	602
29.3.2 Operations	608
29.3.3 Find certain element of the tree	611
29.3.4 Trim the Tree	616
29.3.5 Split the Tree	617
29.4 Exercise	618
29.4.1 Depth	618
29.4.2 Path	619
30 Graph Questions (15%)	623
30.1 Basic BFS and DFS	623
30.1.1 Explicit BFS/DFS	623
30.1.2 Implicit BFS/DFS	623
30.2 Connected Components	625
30.3 Islands and Bridges	629
30.4 NP-hard Problems	631
XI Appendix	635
31 Python Knowledge Base	637
31.1 Python Overview	638
31.1.1 Understanding Object	638
31.1.2 Python Components	639
31.2 Data Types and Operators	642
31.2.1 Arithmetic Operators	642
31.2.2 Assignment Operators	642
31.2.3 Comparison Operators	643
31.2.4 Logical Operators	643
31.2.5 Special Operators	644
31.3 Function	644
31.3.1 Python Built-in Functions	644
31.3.2 Lambda Function	645
31.3.3 Map, Filter and Reduce	646

31.4 Class	647
31.4.1 Special Methods	647
31.4.2 Class Syntax	649
31.4.3 Inheritance	649
31.4.4 Nested Class	649
31.5 Shallow Copy and the deep copy	649
31.5.1 Shallow Copy using Slice Operator	650
31.5.2 Iterables, Generators, and Yield	651
31.5.3 Deep Copy using copy Module	651
31.6 Global Vs nonlocal	652
31.7 Loops	652
31.8 Special Skills	652
31.9 Supplemental Python Tools	654
31.9.1 Re	654
31.9.2 Bitsect	654
31.9.3 collections	654

List of Figures

1	The connection between algorithms	5
1.1	Use category tag to "focus"	16
1.2	Use Test Case to debug	16
1.3	Use Test Case to debug	17
2.1	The State Spaces Graph	22
2.2	The dividing of problems of Divide and Conquer VS Dynamic programming. (Note: the left side in the red box is the Divide and Conquer, and the blue box is the dynamic programming.)	25
2.3	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.	26
3.1	Iteration vs recursion: in recursion, the line denotes the top-down process and the dashed line is the bottom-up process.	33
3.2	Call stack of recursion function	34
4.1	Divide and Conquer Diagram	37
4.2	Two Types of Recurrence Functions	40
5.1	Graphical examples for asymptotic notations. Replace $f(n)$ with $T(n)$	48
5.2	Complexity Chart	50
5.3	The process to construct a recursive tree for $T(n) = 3T(\lfloor n/4 \rfloor) + O(n)$. There are totally $k+1$ levels.	55

5.4	The cheat sheet for time and space complexity with recurrence function. If $T(n) = T(n-1)+T(n-2)+\dots+T(1)+O(n-1) = 3^n$	59
5.5	Complexity of Common Data structures	61
6.1	Two's Complement Binary for Eight-bit Signed Integers.	70
7.1	Array Representation	82
7.2	Linked List Structure	92
7.3	Doubly Linked List	95
7.4	Stack VS Queue	99
7.5	The process of decreasing monotone stack	106
7.6	Example of Hashing Table	113
7.7	The Mapping Relation of Hash Function	114
7.8	Hashtable chaining to resolve the collision	116
8.1	First shows a tree, and how it can be converted to an abstract tree data structure on the right side (put graph here).	125
8.2	Example of a Tree with height and depth denoted	127
8.3	A 6-ary Tree Vs a binary tree.	129
8.4	Example of different types of binary trees	129
9.1	Example of undirected and directed graph: mark the vertices and edges	133
9.2	Bipartite Graph	135
9.3	Four ways of graph representation, renumerate it from 0	139
10.1	Max-heap be visualized with binary tree structure on the left, and implemnted with Array on the right.	144
10.2	Heapify for a given list.	147
11.1	Breath-first search on a simple search tree. At each stage, the node to be expanded next is indicated by a marker.	165
11.2	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.	165
11.3	Binary Tree	170
11.4	The process of Breath-first-search. The black arrows denotes the the relation of u and its not visited neighbors v . All these edges constructs a breath-first-tree. The visiting orders of BFS starting from vertex 0 is $[0, 1, 2, 3, 4, 5, 6]$	174
11.5	Visualize the BFS in level by level fashion.	177
11.6	The search tree using depth-first tree search	180

11.7 The process of Depth-first-search. The black arrows denotes the the relation of u and its not visited neighbors v . And the red arrow marks the backtrack edge.	181
12.1 Example of Rotated Sorted Array	191
12.2 Two pointer Example	198
12.3 Slow-fast pointer to find middle	199
12.4 Floyd's Cycle finding Algorithm	200
12.5 One example to remove cycle	201
12.6 Sliding Window Algorithm	203
12.7 The array and the prefix sum	205
13.1 Tree of possibilities for a typical backtracking algorithm	212
13.2 The state transfer graph of Permutation	214
13.3 The state transfer graph of Combination	217
13.4 All paths from 0, include $0 \rightarrow 1$, $0 \rightarrow 1 \rightarrow 2$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 4$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 3$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$	219
13.5 Example sudoku puzzle and its solution	220
13.6 Example sudoku	221
13.7 An exemplary solution to the eight-queen problem.	226
13.8 Solutions shown of 4×4 chessboard	226
13.9 Caption	227
13.10 Mirroring can cut the search space into half.	230
13.11 Mirroring can cut the search space into half.	230
15.1 Example Graph vs converted tree, where we delete edge $3 -> 5$ and $5 -> 6$	238
15.2 Example of Binary search tree of depth 3 and 8 nodes.	238
15.3 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.	239
15.4 Illustration of Segment Tree.	247
15.5 Trie VS Compact Trie	250
15.6 Trie Structure	250
17.1 Dynamic Programming Chapter Recap	269
17.2 Fibonacci number's Recursion Tree	270
17.3 Subproblem Graph for Fibonacci Sequence till $n=4$	270
17.4 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.	274
17.5 The solution to LIS.	276
17.6 DP Decision	279

17.7 State Transfer for the panlindrom splitting	287
17.8 Summary of different type of dynamic programming problems	303
18.1 Screenshot of Greedy Catalog, showing the frequency and difficulty of this type in the real coding interview	308
19.1 Example of Cycle Detect in both Undirect and Directed Graph	314
19.2 Example of Topological Sort	316
19.3 The connected components in undirected graph, each dashed read circle marks a connected component.	317
19.4 The strongly connected components in directed graph, each dashed read circle marks a strongly connected component.	317
19.5 DF tree is a superset of SCCs. The black dashed block marks a df-tree.	320
19.6 The process of Kosaraju's SCC finding algorithm	321
19.7 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.	322
19.8 A cut denoted with red curve partition V into {1,2,3} and {4,5}.	323
19.9 Prim's Algorithm	323
19.10Prim's Algorithm	326
19.11The process of Kruskal's Algorithm	329
19.12Flattening graph for better observation	333
19.13The execution of Bellman-Ford's Algorithm	335
19.14The process of Dijkstra's Algorithm. At step f, we get the shortest-path tree.	340
19.15The proof of correctness of Dijkstra's Algorithm	340
20.1 The connected components using disjoint set.	350
20.2 A disjoint forest	353
21.1 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.	372
21.2 Word Break with DFS. For the tree, each arrow means check the word = parent-child and then recursively check the result of child.	374
21.3 Caption	379
21.4 One Time Graph Traversal. Different color means different levels of traversal.	386
21.5 Caption	389
21.6 Caption	391
21.7 Tree Structure for One dimensional coordinate	395

21.8 Longest Common Subsequence	397
21.9 Caption	413
22.1 The process of the brute force exact pattern matching	420
22.2 The Skipping Rule	422
22.3 The Sliding Rule	423
22.4 Proof of Lemma	423
22.5 Z function property	428
22.6 Cyclic Shifts	432
22.7 Building a Trie from Patterns	437
23.1 The whole process for insertion sort	444
23.2 One pass for bubble sort	445
23.3 The whole process for Selection sort	447
23.4 Merge Sort: divide process	449
23.5 Merge Sort: merge process	449
23.6 Lomuto's Partition: Three regions in the process, A[p..i], A[i+1..j], A[j+1..e-1]. At position j+1, if A[j+1] <= x, then A[j+1] and A[i+1] is swapped and i and j move forward one location.	452
23.7 The time complexity for common sorting algorithms	459
25.1 Example of floyd's cycle finding	486
26.1 Subsequence Problems Listed on LeetCode	510
26.2 Interval questions	532
26.3 One-dimensional Sweep Line	534
26.4 Min-heap for Sweep Line	534
27.1 Example of insertion in circular list	546
27.2 Histogram	549
27.3 Track the peaks and valleys	555
27.4 profit graph	557
27.5 Task Scheduler, Left is the first step, the right is the one we end up with.	559
28.1 LPS length at each position for palindrome.	566
29.1 Two Cases of K Distance Nodes marked in blue and red ar- rows.	600
29.2 Example of BST to DLL	606
31.1 Copy process	650
31.2 Caption	650
31.3 Caption	651
31.4 Caption	652

List of Tables

1.1	10 Main Categories of Problems on LeetCode, total 877 . . .	13
1.2	Problems categorized by data structure on LeetCode, total 877	15
1.3	10 Main Categories of Problems on LeetCode, total 877 . . .	15
5.1	Explanation of Common Growth Rate	60
7.1	Common Methods for Sequence Data Type in Python	83
7.2	Common Methods for Sequence Data Type in Python	83
7.3	Common Methods of List	85
7.4	Common Methods of String	87
7.5	Common Boolean Methods of String	87
7.6	Methods of Tuple	89
7.7	Common Methods of Deque	103
7.8	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.	104
7.9	Methods for Queue's three classes, here we focus on single-thread background.	104
10.1	Methods of heapq	148
10.2	Private Methods of heapq	150
12.1	Methods of bisect	190
15.1	Time complexity of operations for BST in big O notation . .	245

21.1 Different Type of Single Sequence Dynamic Programming	360
21.2 Different Type of Coordinate Dynamic Programming	360
21.3 Process of using prefix sum for the maximum subarray	366
21.4 Different Type of Coordinate Dynamic Programming	384
31.1 Arithmetic operators in Python	642
31.2 Comparison operators in Python	643
31.3 Logical operators in Python	644
31.4 Identity operators in Python	644
31.5 Membership operators in Python	644
31.6 Special Methods for Object Creation, Destruction, and Rep- resentation	648
31.7 Special Methods for Object Creation, Destruction, and Rep- resentation	649
31.8 Container Data types in collections module.	655

0

Preface

Preface

Interview is the intermediate stage in our life between the long schooling and the real-world employment and contribution. It is a process that literally everyone must go through and trust me it is definitely not an easy and short one. Graduating with a Computer science or engineering degree? Dreaming of getting a job as a software engineer in game-playing companies like Google, Facebook, Amazon, Microsoft, Oracle, LinkedIn, and on and on? While, unluckily, for this type of jobs, the interview process could be the most challengingable among all the interviews because they do "coding interviews", with the interviewer scrutinizing every punch of your typing, while at the same time you need to express whatever that is on your mind to help them get easier to understand what is going on with you. Without passing the "coding interview" you do not even get a chance to discuss about your passion about the company, your passion about the job, or your passion about your life. They just *do not* care before you can approve that you are a competitive "coder".

Normally, how would you prepare for your interview? You Google "How to prepare for the coding interview for A company?" and figured out that you need *Introduction to Algorithms*, *Cracking the Coding Interview* as the basic knowledge base. Then there is the online coding practice website, LeetCode, LintCode. Also, you need to pick a programming language, the inconsistency of all these books and resources that they might use pseudocode or used Java, while you just personally prefer Python. Because why not Python? It is the most popular and easy to use programming language and it would constantly gain more popularity due to the rise of deep learning, machine learning, artificial intelligence. Now, you have a plan in your mind. You pull out your first or second year college textbook *Introduction to Algorithms* from bookshelf and start to read this 1000+-pages massive book to refresh your mind about the basic algorithms, divide and conquer, dynamic programming, greedy algorithm... After this, you go through the

coding interview, or you do both of them at the same time. After that, you would search online to get another book that used Python to handle the common algorithms (do a research of this book). What a tedious, high-intensive, stressful and long process to just prepare for an interview. You would think after this, you are done with the interview, but for software engineers, it is not uncommon to change jobs every two or three years, then you need to start the whole process again unless you are a good recorder that documented your code and everything.

While, good news, this book is written to ease your interview preparation process and provide one-stop and comprehensive knowledge for your "coding interview". I myself is a person who has gone through the whole suffering of interview preparation. We integrate seamlessly of the online resource about interview process, the algorithm concepts from *Introduction to Algorithms*, categorize the real interview problem patterns using LeetCode, the combination of the algorithms and Python language, plus various of concise and hacking algorithms that can solve common interviewing problems with the best complexity which are categorized by the author in order to make the whole picture clearer and the process easier.

We do not simply putting all the contents together, the arrangement of chapter order, the amount of content of each chapter, the connection of each chapter, the explanation of the mindset are based on the LeetCode statistics and real interviewing need.

Exemplary Code and Real Programming in work The code snippets offered in this book, along with the exercises compared with the real world problem solving are toy examples. These toy examples serve a great purpose, to help us understand the properties and behaviors of different algorithms. If we can solve problems on small and more restricted examples, we have laid the foundation to solve the real world and more unrestricted problems.

The purpose of this book We do not simply want a book that teaches you to pass one interview in short time, this is a book that teaches a big range of hands-on data structures, algorithms design methodologies, specific algorithms and more advanced algorithms, and problem-patterns. The book is designed to solve a life-long struggling with coding interviews and try to make you really understand and feel the beauty of programming and algorithms. Utilizing the chance of preparing for your first or dream job to motivate us to really "learn" and be "confident". Is it nice that these algorithm design and problem patterns are one part of your instinct. Before you head off to your interviews, it will not be necessary for you to check out what are the normal problems they would ask about coding interviews.

Our Difference

Compared with other coding interview books, our book is more comprehensive, because we include a large amount of algorithms that can help us excel while not only pass the interviews.

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 algorithms and sense the beauty of them and in the future to use them in your work.

The following diagram based on the connection between different algorithms also represents how we organize our contents:

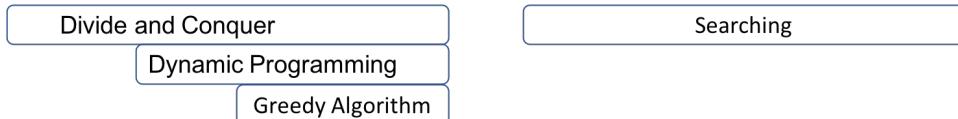


Figure 1: The connection between algorithms

Structure of book

In this book, we first introduce the full interview process to a job, offer a preparation guidance, and current available and helpful online resources to help you prepare; including informational websites, communities, and interview real mocking resource. Second, in our book, each algorithm or data structure chapter is customized based on their frequency, and difficulty appearing in the real coding interviews using statistics from LeetCode, the most popular coding preparation website.

Moreover, compared with a traditional algorithm book, we have extra section which we first categorize all the LeetCode problems on different data structure, from Arrays, to Strings, to Trees, and to Graphs. Then we introduce the commonly used more specific algorithm for these data structures, and we further divide and summarize different type of problems under these data structures and teach you to resolve each type with the algorithms we learned using LeetCode problems.

In this book, we choose to use Python as the programming language due to its popularity and simplicity to use. And we use LeetCode problems either as examples in our book or as exercises after each chapter which makes the book more practical than any other book we have in the market now.

The detailed structure of this book is as follows:

First, because this book is focusing on practical guidance for interviewees to tackle LeetCode systematically. So Part I will introduce the real coding interview, LeetCode, and some resources that might help us smooth your interviewing experience. Also, because this book uses Python as the programming language, so we introduce some important knowledge related to Python that will be needed when you code the solutions to the LeetCode problems. Next, we introduce the methodology used to evaluate the complexity of algorithms.

Next, before we discuss about the structure of this book, let us look at some statistics of the problem tags on the LeetCode website listed on Page [?, p. 150]change the page number). These data structures and type of algorithms are going to be covered in this book, and the amount of information of each section is based on how important that content is. The organization is as:

Part III will discuss about different data structures, and the Chapters are organized in this order: Array and String, Linked List, Graph and Matrix, Hashmap, Stack and Queue, Heap, Binary Tree. In these chapter, each chapter we will focus on the concept introduction and the python implementation. Occasionally some examples and figures will be included to better help us understand. All these chapters would not have too many exercises other than Chapter ???. For example, the binary tree is very special to itself, so that we will include a comprehensive list of type of questions and attack each one, and include large amount of exercise. And for all the other chapter, we

Part ??? will discuss all the basic algorithms, more from the methodology of the algorithms. The Chapters include: Sorting, Divide and Conquer, Dynamic Programming, Searching, Greedy Algorithms, Bit Manipulation, Math and Probability Problems.

And Part ??? are used to complete Part III and Part ??? with more specific algorithms designed for a certain type of questions for each important data structure. For example, there are varied types of algorithms such as two pointers, prefix sum, kadane algorithm and so on for different type of array problems such as subarray, subsequence, sum and so on.

The last Part ??? will offer solutions to all the exercises used in this book.

However, we have extra outline at the end that would have another outline follow the leetcode problems. This, way it would be quicker if you already read the book, and just want to review the questions with certain order.

All the code will be shown in a Jupyter Notebook, and can be downloaded or forked from the github directory.

A reference and simpler content can be found here: <http://interactivepython.org/runestone/static/python.html>

Suggestion of Reading Orders

0.0.1 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.

0.0.2 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.

0.0.3 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:

0.0.4 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.

Due to the Python version

Need to specify which version of python is needed.

Acknowledgements

- A special word of thanks goes to Professor Don Knuth¹ (for T_EX) and Leslie Lamport² (for L^AT_EX).
- I'll also like to thank Gummi³ developers and LaTeXila⁴ development team for their awesome L^AT_EX editors.
- I'm deeply indebted my parents, colleagues and friends for their support and encouragement.

Amber Jain

<http://amberj.devio.us/>

¹<http://www-cs-faculty.stanford.edu/~uno/>

²<http://www.lamport.org/>

³<http://gummi.midnightcoding.org/>

⁴<http://projects.gnome.org/latexila/>

Part I

Introduction

1

Coding Interview and Resources

This chapter is organized as: first, we introduce coding interviews taken by the tech companies in Section 1.1. Next, in Section 1.3, we introduce LeetCode website¹ and its problems. And how we can use this resource to help us with the coding interviews. Also, we provide other resources that can help you smooth your interview experience: including XX, interviewing.io².

1.1 Coding Interviews

For any software engineering position related positions in IoT companies, coding interviews related with data structures and algorithms are necessary. Your masterness of such knowledge varies as the requirement of more specific work.

1.1.1 Process of Interviews

For Interns:

For Full-times:

1.1.2 The Role of Coding Interviews

1.2 Tips and Resources

In this section, we first give tips for the preparation, including the time line and the available resources; then we give more tips of how the real-process

¹<https://leetcode.com/>

²<https://interviewing.io/>

of the coding interviews when you are doing it: either the online with screen sharing or the on-site with whiteboard.

1.2.1 Tips to Preparation

1.2.2 Resources

Online Mocking Interviews

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 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 interviewing.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 community³ that we share information with either interviews, career advice and job packages comparison.

Coding

Geekforgeeks

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.

1.2.3 5 Tips to the Process

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

Tip 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.

Table 1.1: 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%
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%

Tip 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.

³<http://www.1point3acres.com/bbs/>

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 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.

Tips 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.

Tip 4: Master the Chosen Programming Language

1.3 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.

1.3.1 Special Features

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. 1.1. **Use test case to debug** Before we submit our code on the LeetCode, we should use the test case function shown in Fig. 1.3 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

Table 1.2: 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%

Table 1.3: 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%

Topics

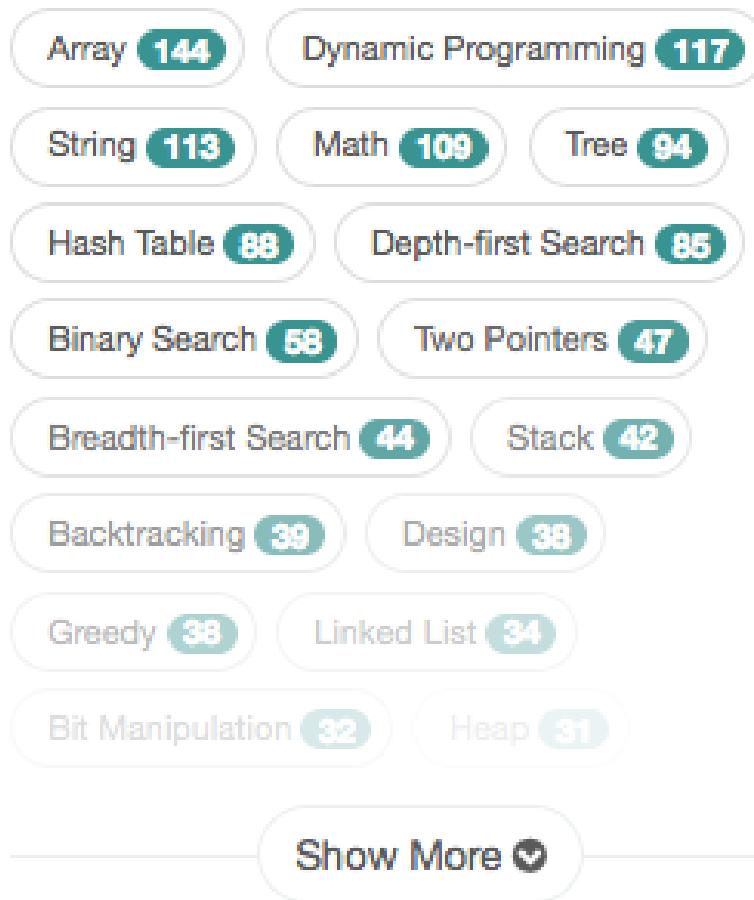


Figure 1.1: Use category tag to "focus"



Figure 1.2: Use Test Case to debug

630. Course Schedule III

A simple C solution 44ms
Created at: May 11, 2018 1:14 AM | No replies yet.

Sort in $n \lg n$, DP using same method as Find Longest Increasing Subsequence
Created at: April 5, 2018 5:15 AM | No replies yet.

Simple Python code using priority queue with explanation
Created at: November 5, 2017 7:36 AM | No replies yet.

C++ priority queue solution
Created at: October 8, 2017 11:42 PM | No replies yet.

Figure 1.3: Use Test Case to debug

2

The Global Picture

“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

I always think the very first of reading any technical book is to build up a global picture which can guide us through the following reading, and see how each part plays its role in the global picture. For algorithm design, it is all about problem modeling and problem solving paradigms.

Problems The fundamental of computer science is to solve problems using programming. The first thing we need to understand should be *problem formulation*. Based on whether the variables are continuous or discrete, we have two categories of problems:

1. Continuous problems.
2. Discrete problems.

We may be asked to answer two types of questions:

1. Find one/all *feasible solutions* that meets problem requirement.
2. Find the *best solution* among all feasible solutions.

If we combine the discrete problems with the above two types of questions, our book is solving these four types of problems:

1. Combinatorial problems: such as permutations, subsets, strings, trees, graphs, points, polygons.

2. Combinatorial optimization problems, aka *discrete optimization*: combinatorial optimization problems are a subset of the combinatorial problems. This subject originally grew out of graph theoretic concerns like edge colorings in undirected graphs and matchings in bipartite graphs. Much of the initial progress is due entirely to theorems in graph theory and their duals. With the advent of linear programming, these methods were applied to problems including assignment, maximal flow, and transportation. In the modern era, combinatorial optimization is useful for the study of algorithms, with special relevance to artificial intelligence, machine learning, and operations research.

Data Structures Throughout this book, we will see that all problems are associated with fundamental data structures such as array, strings, trees, graphs, points, polygons. With different data structure, specific algorithms will be applied to solve these two problems. Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. *In practice, data structures are utilized to model a problem so that it can be solved with a corresponding algorithm.*

Algorithms Algorithms are normally written with programming language, implemented on *data structures*, and are used to seek answers for real-world problems. All combinatorial problems can be solved with basic searching algorithms. This makes *searching* the fundamental strategies and baseline in problem-solving. Searching algorithms will be shown in Part IV. However, the basic searching might not be tractable in the case of complexity, we need other problem-solving strategies namely *divide and conquer*, *dynamic programming* and *greedy algorithms* to make our algorithms tractable.

To summarize it, given a problem description, we need three steps in order to solve it:

1. **Problem-formulation:** In computer science, all problems can be formulated as searching, which has four core elements: *initial state*, *goal state*, *state spaces*, and *state transfer* from one state to another by taking *actions*. And the final result which will either be the sequence of actions to reach to the goal state from the initial state or a set of goal states that satisfy some constraints. Problem formulation is the process of deciding what actions and states to consider, given a goal.
2. **Searching Algorithms:** The process of looking for a sequence of actions that reaches the goal is called search. Therefore, searching is the fundamental algorithm to problem-solving. Equally, we can say all algorithms can be described and categorized as searching algorithms.

- 3. Algorithm Design Paradigms:** Now, we know how we can program and how to reach the goal using naive and straightforward searching algorithms. There are three more general algorithms design paradigms that help us get to the goal faster, they are *Divide and Conquer*, *Dynamic Programming*, and *greedy Algorithms*.

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 3) 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 II), Dynamic Programming and Greedy (Part VI). In Section 2.3, 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 5.6.1.

Data structures and algorithms are inseparable in computer programming.

In order to do comparison between all possible devised algorithms for our problem, we need to learn how to evaluate their performance with time complexity and space complexity. There are some techniques we will introduce before we dive into the four problem solving paradigm and Cracking LeetCode Problems (Part ??), we will learn how to do complexity analysis of algorithms in Section ??.

2.1 Problem-formulation

A problem in computer science can be divided into two categorizes depending on whether the variables are *discrete* or *continuous*. A problem can be defined formally by five components, and we use a simple example to help us explain. Given a list of items $A = [1, 2, 3, 4, 5, 6]$, find the position of item with value 4.

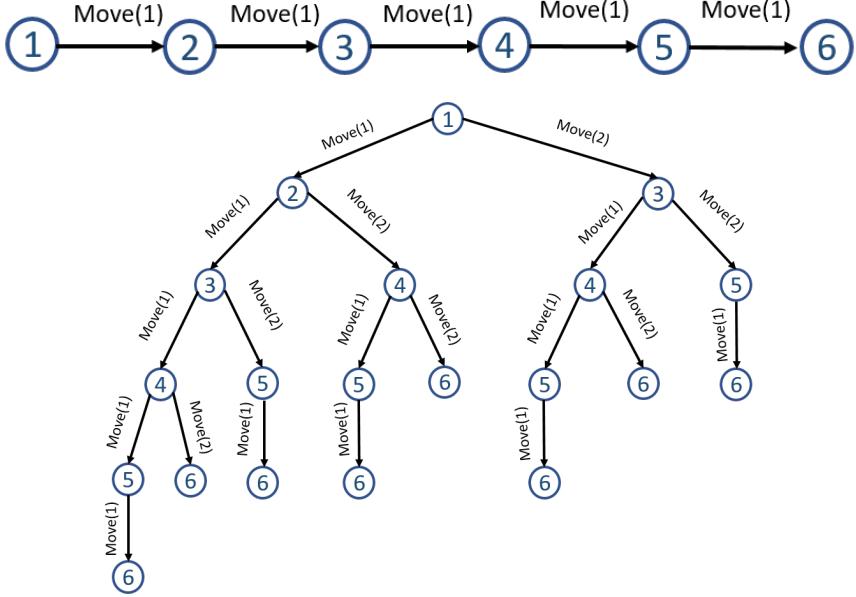


Figure 2.1: The State Spaces Graph

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 $\text{ACTIONS}(1)=\text{MOVE}(1)$, $\text{MOVE}(2)$, $\text{MOVE}(3)$, $\text{MOVE}(4)$, $\text{MOVE}(5)$.
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, $\text{MOVE}(1)$, then we can reach to state 2, denote as $2 = T(\text{MOVE}(1), 1)$. 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. 2.1. 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.

2.2 Searching Algorithms

After the problem formulation, we will have a sense of state spaces, and to reach to the goal state, we need to search a path to reach to this state or a sequence of actions. Searching algorithms will be the most fundamental footstone of all algorithms. Essentially our state spaces are graphs. Under certain limitations, we can simplify it to linear data structure or tree structures.

How searching works? We start from the initial state and form a *search tree* with the initial state as root; the branches are actions and the nodes corresponds to states in the state space of the problem. We search by *expanding* current state; that is, applying each legal action to the current state, thereby reaching to a new set of states. The nodes available for expansion at any given point is called the **frontiers** (also called open list). There are fundamentally three types searching techniques:

1. Breath-first Search: for example, if we are at 1, then our next sets of states are 2, 3. Then we expand the states of 2, then states of 3, and save them to a data structures to expand later, they will be 3, 4, 5. So first, the frontier is 1, then it becomes 2, 3, then at node 2, we expand 3, 4, will end up with 3, 3, 4. This is called breath-first search which expand the frontier in a First-in, first out or FIFO fashion (FIFO queue). Breath-first search is usually implemented iteratively.
2. Depth-first Search: similarly at state 1, we get our frontiers to be 2, 3. But if we deal with the nodes in the frontier in a last-in, first out or LIFO fashion (LIFO queue and also known as stack). At node 1, our generated frontier will be [2, 3], then we go to state 3, and expand more states here, the resulting frontier will be [2, 4, 5]. Then move to state 5, with resulting frontier [2, 4, 6]. Depth-first search can either be implemented iteratively or recursively.
3. Priority based Search: if we pop the elements of the queue with the highest priority according to some ordering function, this is called propriety based search, this can be implemented with a priority queue.

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 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.

2.3 Algorithm Design Paradigms

To solve a real-word problem, we first read the description thoroughly to frame the problem into a programmable way: the input data structure, the output result, and an algorithm that can take the input and get the output with its logic.

More of the time, the most naive and inefficient solution – *brute-force solution* would strike us right away, which is simply imitating the problems so that it will be solved by the computer utilizing its massive computation power. 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 general, there are four algorithm design paradigms for reference when designing your more competitive solution, which in this section we will briefly offer a glimpse into their concepts and discerns. They are:

1. Divide and Conquer
2. Dynamic Programming
3. Greedy Algorithm

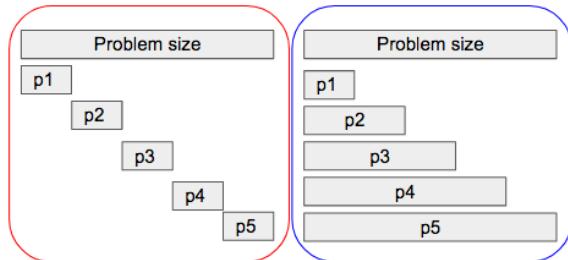


Figure 2.2: The dividing of problems of Divide and Conquer VS Dynamic programming. (Note: the left side in the red box is the Divide and Conquer, and the blue box is the dynamic programming.)

Divide and Conquer is the most fundamental programming philosophy. It first recursively break a problem into smaller non-overlapping subproblems till a small base subproblem which can be solved easily, and then combining the results of the subproblems into the solution to its superproblem in some way. The process is demonstrated in Fig 2.2, and it usually be implemented with recursive function. It usually decrease the time complexity of logarithm level. For instance, it optimize a $O(n^2)$ time complexity to $O(n \log n)$.

Dynamic programming follows the same philosophy of Divide and Conquer and commonly used to tackle optimization problems. It also first break a problem into subproblems. But instead of the non-overlapping subproblems, their subproblems overlaps in a way as demonstrated in Fig 2.2, which means a larger size subproblem grows from smaller previous subproblems. The solution to current subproblem can depends on any number of previous smaller subproblems. With dynamic programming, intermediate results are cached and can be used in subsequent operations.

Greedy algorithms often involve optimization and combinatorial problems; the classic example is applying it to the traveling salesperson problem, where a greedy approach always chooses the closest destination first. This

shortest path strategy involves finding the best solution to a local problem in the hope that this will lead to a global solution.

Complete Search Complete search, also known as brute force or recursive backtracking, is a method for approaching a problem by naively searching through the whole solution spaces to obtain the required solution. Optimization is through pruning the searching space by ending invalid searching early. Complete search is used when there is clearly no clever algorithms available (algorithms that use one of previous three paradigms), for instance with permutation and combination stated in Section 13.1.2, or when such clever algorithms exist, but overkill when the input size happen to be small for complete search.

On the LeetCode, a lot of times, complete search should be the first considered solution that come to mind. With bug-free complete search solution, we should never receive Wrong Answer response, but we might get Time Limited Error (TLE) instead due to its high time complexity.

2.4 Problem Modeling

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

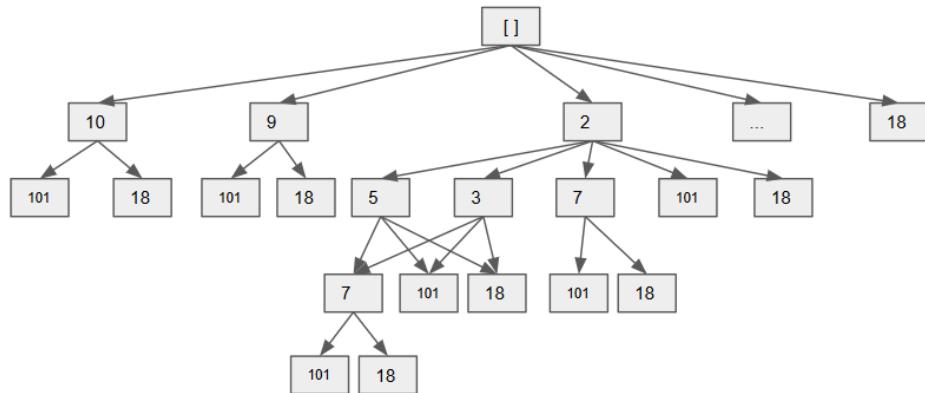


Figure 2.3: State Transfer Tree Structure for LIS, each path represents a possible solution. Each arrow represents a move: find an element in the following elements that's larger than the current node.

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 because 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: $\text{nums}[i+1]$, $\text{nums}[i+2]$, ..., $\text{nums}[n-1]$. 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.

The above example is to show us, how learning and practice using the data structures and four problem solving paradigms can help us making smarter decision about problem modeling and problem solving.

Part II

Fundamental Algorithm Design and Analysis

This purpose of this part is to embody readers of the fundamental algorithm design methods before we head off to data structures, advanced algorithm design, and LeetCode problem solving.

We include three chapters: Iteration and Recursion (Chapter 3), Divide and Conquer (chapter 4, and Algorithm Analysis (chapter 5).

Iteration and recursion are key computer science techniques used in creating algorithms and developing softwares. Iteration is easy to understand, therefore, we focus more about understanding recursion, and list common usages of recursion.

Divide and conquer is the core algorithm design method, which mostly goes hand in hand with recursion.

And algorithm analysis methods usually varies upon either it is iteration or recursion. The analysis of iteration is trivial compared with the one with recursion. In Chapter, we mainly focus on the main three methods developed to analysis the complexity of recursion.

3

Iteration and Recursion

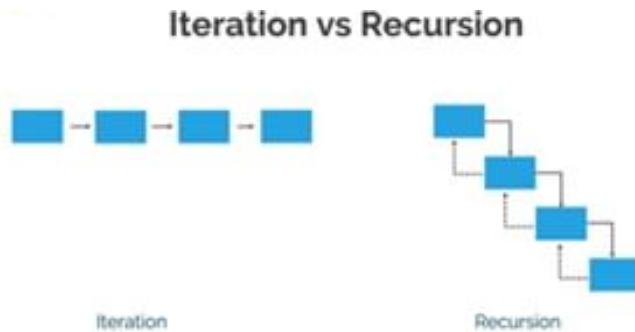


Figure 3.1: Iteration vs recursion: in recursion, the line denotes the top-down process and the dashed line is the bottom-up process.

The purpose of this chapter is to understand how iteration and recursion works in algorithms and software developing. Especially, to understand how recursion works, and we also include the most common usage of being recursion.

3.1 Iteration VS Recursion

In simple terms, an iterative function is one that loops to repeat some part of the code, and a recursive function is one that calls itself again to repeat the code. Using a simple for loop to display the numbers from one to ten is an iterative process. Examples of simple recursive processes aren't easy to find, we have tree traversal, depth-first-search, implementation of divide and conquer.

3.2 Recursion

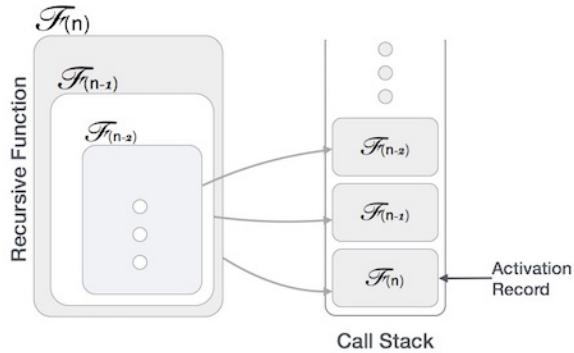


Figure 3.2: Call stack of recursion function

Different from iteration, recursion function uses a stack to record the recursive function calls, and it needs to return from the base case to the upmost level of function call. This feature makes it possible for recursion to handle a problem in two directions: top-down and bottom-up. In real problem solving, different process normally has different usage.

In top-down process we do:

1. Break problems into smaller problems: non-overlapping(Divide and conquer) and overlapping(Dynamic programming).
2. Iteration: 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:

1. return None: Simply return to the upper level
2. return variables: if we have return result, we do process of these results with current state and return to the upper level. In divide and conquer, we mostly likely need to merge its results. For iteration, this process gives the iteration process the backward traveling process.

For iteration, the top-down process is visiting nodes in 'forwarding' direction, and the bottom-up process on the other hand functions as a reverse visiting process. This makes a linear data structures function as a doubly linked list, and make a one direction tree structure function as one with parent. Here we list some examples that used recursive so that we can go backward:

1. 2. Add Two Numbers

Overflow problem

3.3 Summary

A conditional statement decides the termination of recursion while a control variable's value decide the termination of the iteration statement (except in the case of a while loop). Infinite recursion can lead to system crash whereas, infinite iteration consumes CPU cycles. Recursion repeatedly invokes the mechanism, and consequently the overhead, of method calls. This can be expensive in both processor time and memory space while iteration doesn't. Recursion makes code smaller while iteration makes it longer.

4

Divide and Conquer

“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

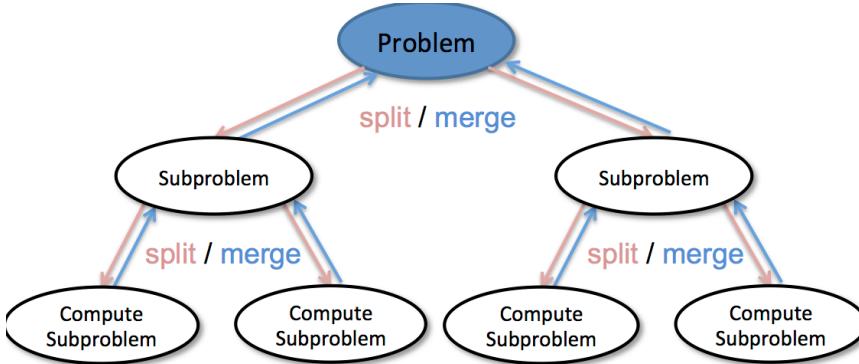


Figure 4.1: Divide and Conquer Diagram

In this chapter, we learn our the first and the most fundamental paradigm of the four we introduced – *Divide and Conquer*. *Recursion* in most programming languages are a recursive function that calls itself and return till it hits its *base cases*. Divide-Conquer and Recursion are inseparable with each other, as shown in Fig 4.1, with divide and conquer, a problem is break down to two subproblems, recursively, till we reach to the base cases

where there solutions is trivial to solve. The relation between the problems and its subproblems can be represented with *Recurrence Function* such as $T(n) = 2 * T(n/2) + f(n)$, where for the original problem of size n , it was break into 2 subproblems each has a size $n/2$, and $f(n)$ is the cost function denotes the time complexity it takes to merge the result of the two subproblems back to the solution of the current problem of size n . Therefore, recurrence function and a tree structure is a nature way for us to either visualize the process or deduct the time complexity.

4.1 Dividing and Recurrence Function

For divide and conquer, we divide a problem into a number of subproblems recursively. Generally, we use **recurrence function** to describe the relations between a problem and its subproblems. Depends on how the problem is divided, it can results two type of subproblems: *non-overlapping* and *overlapping*. Optionally, we can also use **recursion tree** instead of recurrence function to describe their relation which we will further this subject in more details later.

The recurrence function is popularized to be used to describe the running time of algorithms fall into divide-and-conquer fashion due to its convenience and succinctness to characterize the relations. In a recurrence function, we use $T(n)$ to denote the function which represents the ‘time’ cost to a problem of size n . Another subpart of this equation is $f(n)$ which represents the cost of dividing a problem into subproblems and combining them back to its upper level problem. For convenience, we can use $D(n)$ as the running time to divide the problem of size n into a subproblems each with size n/b , and $C(n)$ be the cost combines the solution of these subproblems into the solution of its upper level problem. Consequentially, $f(n) = D(n) + C(n)$. In reality, $D(n)$ is trivial compared with the cost of combining step. We use $f(n)$ in our book for simplicity purpose.

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, we can use Eq.4.1 to denote this relation.

$$T(n) = a * T(n/b) + f(n) \quad (4.1)$$

where $a \leq 1, b > 1$. In the equal dividing, a problem of size n is divided into a subproblems each of size n/b . In a recursion manner, the problem $T(n/b)$ will further be divided into $T(n/b^2)$, and $T(n/b^3)$ up till step k where the subproblem of size n/b^k is small enough to be solved directly.

To better describe the non-overlapping feature, we can represent the cost of dividing and combing $f(n)$ with a nested function P , which denotes

a problem which has two parameters, starting index and ending index in the problem array. The original problem is denoted as $P(0, n)$, the relation of $f(n)$ is further detailed in Eq. 4.2.

$$f(P(0, n)) = f(P(0, n/b), P(n/b + 1, 2n/b), \dots, P(n - n/b, n)) \quad (4.2)$$

which captures the relation between problem and all corresponding subproblems and highlight the non-overlapping features.

Overlapping subproblems Different from the non-overlapping problems, the feature of overlapping problem is more abstract. A easy way to understand is to visualize it. where our subproblems has overlapping elements.

$$T(n) = T(n - 1) + T(n - 2) + \dots + T(1) + f(n) \quad (4.3)$$

Or in Recursion calls,

$$F(n) = \text{Comb}(F(n - 1), F(n - 2), \dots, F(1)) \quad (4.4)$$

We can have any combination of terms $T(k)$, $k = [1, n - 1]$ on the right side of this recurrence equation. This equation denotes the problem of size n is divided into subproblems each with possible size from 1 to $n-1$, and where subproblem $n-1$ overlaps with subproblem $n-2$, and so on.

Example 1: 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)

```

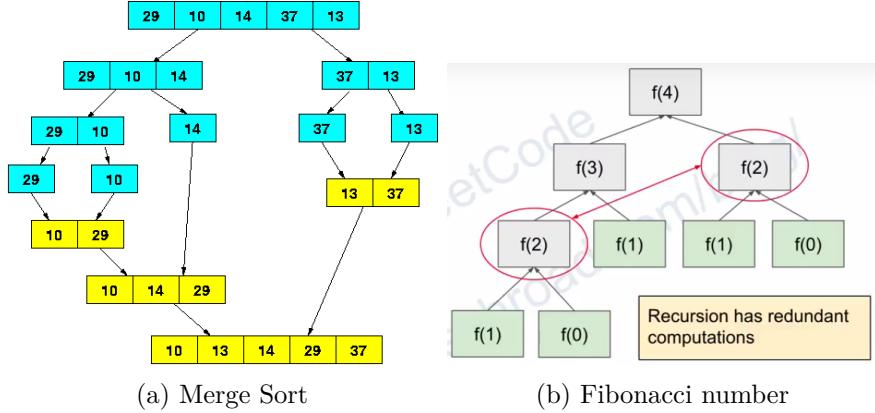


Figure 4.2: Two Types of Recurrence Functions

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 4.2b. And we also draw the recursion tree of recursion function call for merge sort and shown in Fig 4.2a. 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 $T(n) = T(n - 1) + T(n - 2) + 1$.

Divide-and-conquer VS Dynamic Programming Therefore, we draw the conclusion:

1. For non-overlapping problems as Eq. 4.1, 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. 4.3, 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 tablization as we later explain in Chapter **Dynamic Programming** (Chapter 17).

In the following book, we use divide and conquer to refer to the first category of dividing a problem into non-overlapping subproblems as Eq. 4.1. Follow this, we give the official definition of Divide-and-conquer.

4.2 Divide and Conquer

Definition Divide and conquer is the most fundamental problem solving paradigm for computer programming. Divide and conquer solves a given problem recursively, the recursion function is composed of three parts:

1. **Divide:** divide one problems into a series of non-overlapping subproblems that are smaller instances of the same problem until reaching to the *bases cases* where the subproblem is trivial to solve – usually by half and half.
2. **Conquer:** solve the subproblem by calling the function itself (recursively) with parameters represent its corresponding subproblem and return its solution.
3. **Combine:** combine the solution from each subproblem into the solution to the current problem.

Applications 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 23);
- Binary Search (Section 12.1);
- Heap(Section 10.1);
- Binary Search Tree (Section 15.1);
- Segment Tree(Section 15.2).

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.

4.3 More Examples

- 4.1 **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):
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(
28         low ,mid) #[low ,mid]
29         rslt.append(maxSubarray(low ,mid)) #[low ,mid]
30         #right_low ,right_high ,right_sum = maxSubarray(
31         mid+1,high )#[mid+1,high ]
32         rslt.append(maxSubarray(mid+1,high ))
33         #cross_low ,cross_high ,cross_sum = getCrossMax(
34         low , 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]

```

Also, we does not necessarily to use divide and conquer, we can be more creative and try harder to make the time complexity goes to $O(n)$. We can convert this problem to best time to buy and sell stock problem.[0, -2, -1, -4, 0, -1, 1, 2, -3, 1], => 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

```


5

Complexity Analysis

Complexity analysis is a tool for us to evaluate our algorithm's performance beforehand of the actually running on the hardware. Resources such as memory, communication bandwidth, or computer hardware are of primary concern. But, the most important measurement we want is the computational complexity, to measure the time that needed to run a specific algorithm with a sized input. In algorithm analysis, these two type of complexity of algorithms are referred as space and running time complexity.

In this chapter, we centralize on the techniques to analyze the complexity – both time and space complexity – of an algorithm or a data structure. Both time and space are resources and cost to pay to get the programming result. Before we dive into a plethora of algorithms and data structures, learning the complexity analysis techniques can ease our understanding and evaluating our each algorithm. The ordering of this chapter is:

1. Asymptotic Notations (Sec. 5.1): introduce three cases of notations used both in time and space complexity analysis, include best-case, worst-case, and average-case analysis. Understanding that the same algorithm being applied on different input data can result different time complexity.
2. Three Cases Time Complexity Analysis (Sec. 5.2): We introduce how to perform time complexity Thfirst for simple iteration programming, then to analyze the recurrent function of: divide-conquer and dynamic programming.
3. Amortized Analysis (Sec. 5.4).

4. Space Complexity (Sec. ??): space complexity is usually way more straightforward compared with time complexity.
5. Time complexity VS space complexity (Sec. ??).

Complexity Analysis in Coding Interviews No one can escape complexity analysis in real coding interviews just like nearly no one can escape the gravity of black holes. In reality, you always start with the naive solution and most likely to offer either average-case or worst-case time complexity. Then with leverage of another data structure or algorithm, you make gradual improvement on the previous version. This process shows how you approach a given problem and choose the one you think fit the best evaluated with complexity analysis.

In order to analyze and answer the time complexity, we need to (1) the right way to denote the time complexity, which are three asymptotic running time. (2) For recursive programming, come up with its right recurrence time complexity function and master techniques to solve the function and get a proper asymptotic running time. (3) In order to answer these questions quickly since it is a short but import part, we need to memorize the most commonly used algorithms' time complexity. We provide a cheatsheet at the end of this chapter.

Recurrence Function All the algorithms can be represented by a Recurrence Function, solving this recurrence function will get us the time complexity.

5.1 Notations and Definitions

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 $T(m, n)$.

Let us see an example with selection sort (details in Chapter 23). In selection sort, each time it selects the current largest item and swap it with item at its right position. Given the input array A , and size to be n , we have index $[0, n - 1]$. At the first pass, we choose the largest item from $A[0, n - 1]$

and swap it with $A[n - 1]$. At the second pass, we choose the largest item from $A[0, n - 2]$ and swap it with $A[n - 2]$. Totally, after $n - 1$ passes we will get an incrementally sorted array. 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
6         li = 0
7         for j in range(n - i):
8             if a[j] > a[li]:
9                 li = j
10    # swap li and ti
11    print('swap', a[li], a[ti])
12    a[ti], a[li] = a[li], a[ti]
13    print(a)
14 return a

```

A simple mathematical operation takes constant time c . In the above code, the line that comes with cost and items notations are operations. In line 5, we first point our 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 $\sum_{i=0}^{n-2}(n - i)$ because of there are 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 .

$$\begin{aligned}
T(n) &= 3c * (n - 1) + \sum_{i=0}^{n-2} 2c(n - i) \\
&= 3c * (n - 1) + 2c(n + (n - 1) + (n - 2) + \dots + 2) \\
&= 3c * (n - 1) + 2c\left(\frac{(n - 1) * (2 + n)}{2}\right) \\
&= cn^2 + cn - 2 + 3cn - 3c \\
&= cn^2 + 4cn - 3c - 2 \\
&= an^2 + bn + c
\end{aligned} \tag{5.1}$$

The equation 5.1 is unnecessarily complex to represent the order of complexity. We introduce **asymptotic notations** in the next section to see how it can be simplified.

Asymptotic Notations

Order of Growth and Asymptotic Running Time In Equation 5.1 we end up with three constant a , b , c and two types two order terms n^2 and n . When the input is large enough, all the lower order terms will become

relatively insignificant even if with large constant; we thus neglect the lower terms. Further, we neglect the constant coefficient a for the same reason. Therefore, for equation 5.1, we say $T(n) = n^2$ instead. The relation between the original function $an^2 + bn + c$ and n^2 is **asymptotic**, and denoted as $an^2 + bn + c \asymp n^2$. For example, if the input size is n , then we can have functions like $f_1 = an + b$ and $f_2 = an^2 + bn + c$. But normally, we would only get the highest order of function, and simplified them to $g_1 = n$ and $g_2 = n^2$, with different notation. Function f_1 and g_1 are of the same order of magnitude or growth and g_1 can approach the curve of f_1 arbitrarily closely. The relation of them is **asymptotic**, and denoted as $f_1 \asymp g_1$.

Asymptotic Notation Asymptotic notations can be applied to characterize both running time complexity and space complexity. We focus on running time instead due to space complexity's simplicity in the analysis compared with the running time. To analyze the complexity of an algorithm in each space or time, we have three ways (shown in Fig. 5.1):

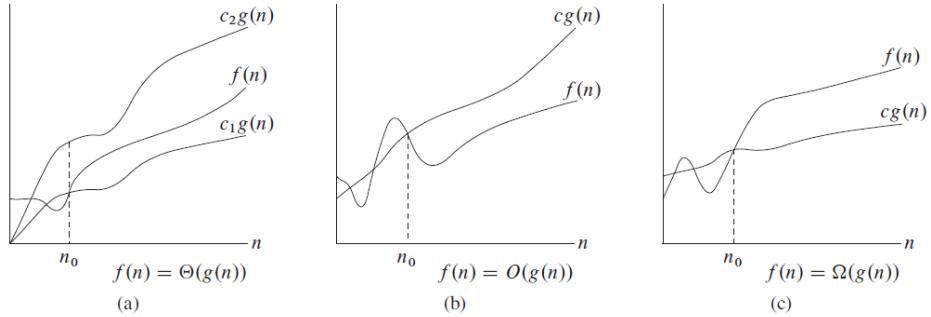


Figure 5.1: Graphical examples for asymptotic notations. Replace $f(n)$ with $T(n)$

1. **Θ -Notation:** We denote the **asymptotic tight bound** running time as $T(n) = \Theta(g(n))$ (pronounced as “big theta”), which denote **a set of functions** that 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 . $\Theta(g(n)) = \{f(n)\}$. We say $g(n)$ is asymptotically tight bound for $T(n)$. 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 $T(n) = \Theta(n^2)$.
2. **O -Notation:** The **asymptotic upper bound** running time as $T(n) = O(g(n))$ (pronounced as “big oh”), we denote the set of functions $0 \leq f(n) \leq c g(n)$ for all $n \geq n_0$ for positive constant c and n_0 . Note that $T(n) = \Theta(g(n))$ also implies that $T(n) = O(g(n))$. With our example,

it can be also $T(n) = O(n^2)$. To get the upper bound, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure, which gives out the worst-case running time.

3. **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 .

We should note that only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, we can have $f(n) = \Theta(g(n))$. With this theorem, we know that n^2 is both upper and lower bound for selection sort algorithm. Also, the equality relation really means “belong to”, $T(n) \in O(g(n))$ and the same for the others.

Four Types of Complexity Analysis

As we will see in the remaining content of the book, different algorithm is affected differently with its input data distribution. We category this influence in three levels: worst-case, average-case, and best case.

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.

The selection algorithm's complexity will not be affected by its input, because no matter what data distribution is, we need to traverse the fixed range of array to find the largest item. For these type of algorithms, usually we have $T(n) = O(g(n)) = \Theta(g(n)) = \Omega(g(n))$. However, there are a lot of other algorithms that differs from different inputs.

All of these notations are applied to functions. However, in the practical interviews, when the interviewer asks you to give the time and space complexity, you do not necessarily have to give them the answer for each notation, you can just use O to denote, with regard to different cases introduced in the next section. Here, we provide the most likely used growth rate plotted in Fig. 5.2.

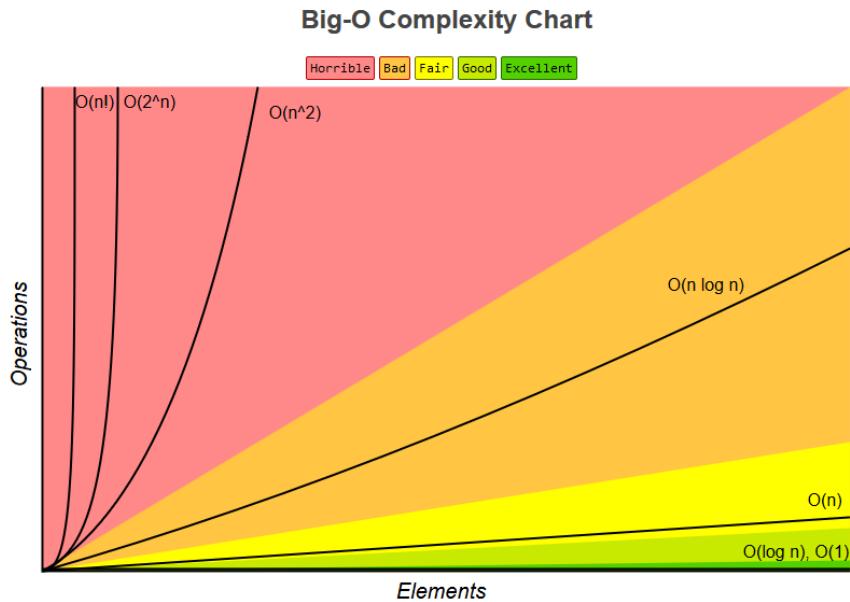


Figure 5.2: Complexity Chart

The reality, worst-case and average is more often used compared to best-case running time analysis. There are two different ways to evaluate an algorithm/data structure:

1. Consider each operation separately: one that looks at 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.

5.2 Three Cases Time Complexity Analysis

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     s1 = [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            sorted element
13            s1[j+1] = s1[j] # shift a[j] backward
14            j -= 1
15        s1[j+1] = key
16        print(s1)
17    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:

$$\begin{aligned} T(n) &= T(n - 1) + O(n) \\ &= O(n^2) \end{aligned} \tag{5.2}$$

And,

$$\begin{aligned} T(n) &= T(n - 1) + \Omega(1) \\ &= \Omega(n) \end{aligned} \tag{5.3}$$

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:

$$\begin{aligned} T(n) &= T(n - 1) + \Theta(n/2) \\ &= T(n - 2) + \Theta(\frac{n}{2} + \frac{n-1}{2}) \\ &= \Theta(n^2) \end{aligned} \tag{5.4}$$

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.

5.3 Solve Recurrence Function

In the above example, it is the time analysis for linear programming. For recursive programming, it is more complex and as we have discussed in previous Chapter 3, we can use recurrence function to denote the $T(n)$, which we have two types:

1. **Non-overlapping Recurrence Function:** as $T(n) = aT(n/b) + f(n)$, where $f(n)$ can be replaced by the right asymptotic notation. This is the form of complexity function using divide and conquer methodology.
2. **Over-lapping Recurrence Function:** which has way more cases, such as $T(n) = T(n - 1) + T(n - 2) + f(n)$.

$f(n)$ denotes the cost that convert a problem of size n to another subproblem. We can use any of the asymptotic notations introduced above to decide the cost of "dividing". To get the eventual asymptotic running time we

need to solve these tricky recurrence time function. Luckily, there are a few techniques for converting recursive relations to closed formulas. Solving a recurrence relation means we need to find a function of n (a closed formula) which satisfies the recurrence function, as well as the initial condition. Checking the correctness of a solution is quite easy compared with finding the solution.

1. **Iteration and Recursion Tree:** when the iteration relation is simple, meaning $T(n)$ is only related to another item such as $T(n/b)$ or $(T(n-1), T(n-2))$, we can simply expand the iteration function from $T(n)$ to $T(1)$. From the expansion, using summing over iteration, we can get the closed formula for $T(n)$. When the relation is more complex, we can iterate and expand with recursion tree for better visualization.
2. **Guess with the characteristic root technique:** We observe some rules and guess the final closed formula with some parameters to be decided later. We replace the recursion with our guessed formula, and together with the initial condition, we can get a final result.
3. **Master Method:** For the non-overlapping function, people have generalized the rule of the closed formula which we can just memorized it and apply it when computing for time complexity.
4. **Wild Guess + Induction proof:** After we have enough experience with recursion relation, we can just make a guess of the final closed formula and prove it with induction method.

Lets explain each method one by one with practical examples. The example of insertion sort we used the iteration method to expand $T(n)$. Let us apply it on recursion: $T(n) = T(n/2) + O(1)$ (can be seen in binary search algorithm), $T(n) = T(n-1) + T(n-2) + O(1)$ (such as Fibonacci Sequence), and $T(n) = 3T(n/4) + O(n)$. Before we head off we first introduce some math formula to help us ease the process:

$$t^0 + t^1 + \dots + t^n = \frac{1 - t^{n+1}}{1 - t} \quad (5.5)$$

$$a^{\log_b n} = n^{\log_b a} \quad (5.6)$$

5.3.1 Iteration and Recursion Tree

We demonstrate iteration with a simple non-overlapping recursion.

$$T(n) = T(n/2) + O(1) \quad (5.7)$$

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

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

$$= \dots$$

$$= T(1) + kO(1) \quad (5.8)$$

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.

$$T(n) = 3T(n/4) + O(n) \quad (5.9)$$

$$= 3(3T(n/4^2) + n/4) + n = 3^2T(n/4^2) + n(1 + 3/4)$$

$$= 3^2(3T(n/4^3) + n/4^2) + n(1 + 3/4) = 3^3T(n/4^3) + n(1 + 3/4 + 3/4^2) \quad (5.10)$$

$$= \dots \quad (5.11)$$

$$= 3^kT(n/4^k) + n \sum_{i=0}^{k-1} \left(\frac{3}{4}\right)^i \quad (5.12)$$

Since the term of $T(n)$ grows, the iteration can look messy. We can use recursion tree to better visualize the process in Fig. 5.3. First we only have $T(n)$ term, and we expand $T(n) = L_1 + L_2$. Then expand further $T(n) = L_1 + L_2 + L_3$. After level 3, we should try to generalize the representation of a level. In this example, the generalization is shown in the dashed red box. We can also observe that there will be $k+1$ levels, where the first k levels will only be related to $f(n)$ and has nothing to do with $T(n)$ and the $k+1$ -th level will only be related with $T(n)$. In this case, it is the base case $T(1)$. Through the expansion with iteration and recursion tree, our time complexity function becomes:

$$T(n) = \sum_{i=1}^k L_i + L_{k+1} \quad (5.13)$$

$$= n \sum_{i=1}^k (3/4)^{i-1} + 3^k T(n/4^k) \quad (5.14)$$

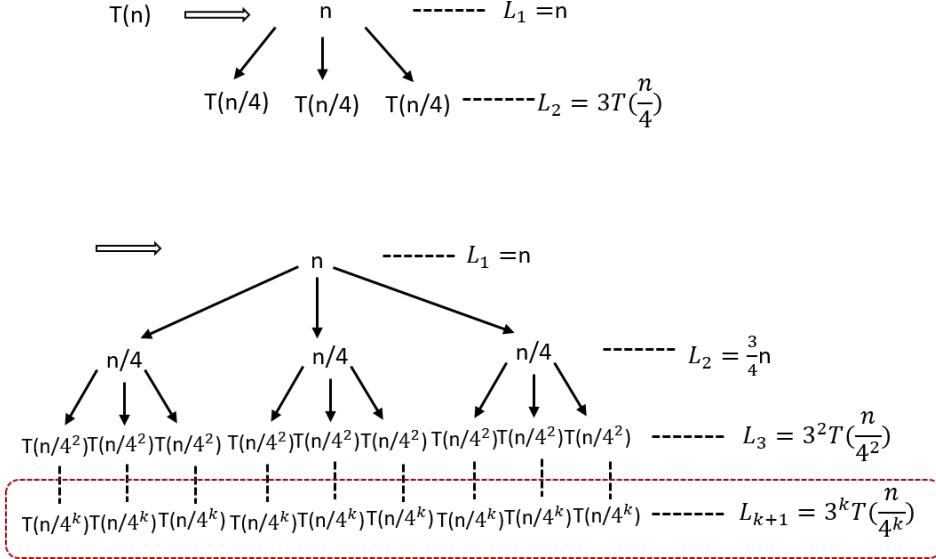


Figure 5.3: The process to construct a recursive tree for $T(n) = 3T(\lfloor n/4 \rfloor) + O(n)$. There are totally $k+1$ levels.

In the process, we can see that Eq. 5.15 and Eq. 5.9 are the same. Because $T(n/4^k) = T(1) = 1$, we have $k = \log_4 n$.

$$T(n) \leq n \sum_{i=1}^{\infty} (3/4)^{k-1} + 3^k T(n/4^k) \quad (5.15)$$

$$\leq 1/(1 - 3/4)n + 3^{\log_4 n} T(1) = 4n + n^{\log_4 3} \leq 5n \quad (5.16)$$

$$= O(n) \quad (5.17)$$

5.3.2 Characteristic Root Technique

In this section, we use the overlapping subproblem related complexity function to show how we use recursion tree and characteristic root technique to get the same time complexity.

$$T(n) = T(n-1) + T(n-2) + O(1) \quad (5.18)$$

$$\begin{aligned} &= (T(n-2) + T(n-3) + O(1)) + (T(n-3) + T(n-4) + O(1)) + O(1) \\ &= \dots \end{aligned} \quad (5.19)$$

It is actually very difficult for us to keep expanding, because the item of $T(k)$ will double each time we try to expand it, this is eventually grow exponentially. If we insist on expanding, we can use recursion tree method. We can get a tighter bound using recursion tree method, where instead we just use L_k to represent each level and we can do simple merge of different

term, which is easier for us to find rules:

$$T(n) = T(n - 1) + T(n - 2) + O(1) \quad (5.20)$$

$$L_1 = \underline{T(n - 1)} + \overbrace{T(n - 2)} + O(1) \quad (5.21)$$

$$L_2 = \overbrace{(T(n - 2) + 2T(n - 3) + T(n - 4))} + O(2^1)$$

$$L_3 = \overbrace{(T(n - 3) + 3T(n - 4) + 3T(n - 5) + T(n - 6))} + O(2^2)$$

$$L_4 = \overbrace{(T(n - 4) + 4T(n - 5) + 6T(n - 6) + 4T(n - 7) + T(n - 8))} + O(2^3)$$

$$L_k = \underbrace{(T(n - k) + \dots + T(n - k - k))}_{T(c) \text{ Reminder}} + \underbrace{O(2^{k-1})}_{\text{Summation of } f(n) \text{ at each level}}$$

The last level we will have $T(n - 2k) = T(1) = 1$, which states $k = n/2$. For the final time complexity, it will be composed of two parts: (1) the summation of the $f(n)$ part over different level, which is $\sum_{i=0}^{k-1} 2^i$; and (2) the remainder of the $T(c)$, which states the cost for trivial case. It is $(T(n - n/2) + \dots + T(1))$, and according to the rule of the coefficient, $(T(n - n/2) + \dots + T(1)) \geq 2^k$. With the math formula that $t^0 + t^1 + \dots + t^n = \frac{1-t^{n+1}}{1-t}$. Therefore, we can rewrite our time complexity function as:

$$T(n) \geq \sum_{i=0}^{k-1} 2^i + 2^k \quad (5.22)$$

$$= \sum_{i=0}^k 2^k \quad (5.23)$$

$$= \frac{1 - 2^{k+1}}{1 - 2} \quad (5.24)$$

$$= 2^{k+1} \quad (5.25)$$

$$= 2^{n/2} \quad (5.26)$$

It would be reasonable for us to guess r^n . For this type of recursion, it is hard to find the tight bound, we can do the following simplification to find a lower bound instead, and replace each term with out guess

$$T(n) \geq T(n - 1) + T(n - 2) \quad (5.27)$$

$$r^n \geq r^{(n-1)} + r^{(n-2)}$$

$$r^{(n-2)}(r^2 - r - 1) \geq 0 \quad (5.28)$$

With some math knowledge that given a general quadratic formula as $ax^2 + bx + c = 0$, the solution will be $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. With the formula, we get

the solution for equation $(r^2 - r - 1) = 0$, which is $r = \frac{1 \pm \sqrt{5}}{2}$:

$$T(n) \geq A\left(\frac{1 + \sqrt{5}}{2}\right)^n + B\left(\frac{1 - \sqrt{5}}{2}\right)^n \quad (5.29)$$

$$T(n) \geq A\left(\frac{1 + \sqrt{5}}{2}\right)^n \quad (5.30)$$

$$T(n) \geq 2^{n/2} \quad (5.31)$$

$$T(n) = \Omega(2^{n/2}) \quad (5.32)$$

5.3.3 Master Method

Recursion tree and the master theorem are the main ways we rely on to answer the time complexity for a divide and conquer method of form shown in Eq. 5.33. The master method is probably the easiest way to come up with the computational complexity analysis. It is a theorem that are proved by researchers, and we just need to learn how to use them. The master theorem goes:

$$T(n) = aT(n/b) + f(n) \quad (5.33)$$

For Eq. 5.33, let $a \geq 1, b > 1$, and $f(n)$ is asymptotically positive function. This represents that using divide and conquer, we divide a problem of size n into a subproblems and each of size n/b . The a subproblems are solved recursively, each in time $T(n/b)$. Plus with the cost of $f(n)$, which represents the cost of dividing the problem and combining results of the subproblems, we get the time complexity of size n .

The master theorem states that for Eq. 5.33, $T(n)$ has three following asymptotic bounds.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for constant $\epsilon > 0$, then we get $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then we get $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for constant $c < 1$ and all sufficiently large n , then we get $T(n) = \Theta(f(n))$.

Apply Master Method To apply master method given a function $T(n)$, we first compute $n^{\log_b a}$, and then compare $f(n)$ with $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. As shown in the following equation of case 1 and 3. For case 2, these two functions are of the same size, we multiply it by a logarithmic factor.

$$f(n) > n^{\log_b a}, \text{ case 3, } T(n) = \Theta(f(n)) \quad (5.34)$$

$$f(n) < n^{\log_b a}, \text{ case 1, } T(n) = \Theta(n^{\log_b a}) \quad (5.35)$$

$$f(n) = n^{\log_b a}, af(n/b) \leq cf(n), \text{ case 2, } T(n) = \Theta(n^{\log_b a} \log n) \quad (5.36)$$

Note that the comparison will be polynomial comparison.

When We cant Use Master Method The three cases do not cover all the possibilities for $f(n)$. There is a gap between case 1 and 2 when $f(n)$ is smaller but not polynomially smaller. Similarly, there is a gap between case 3 and 2 when $f(n)$ is larger but not polynomially larger. Or if the regularity condition in case 3 fails to hold, we can not use master method. We go back to other techniques instead.

5.4 Amortized Analysis

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 view 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:

5.5 Space Complexity

The space the recursive function occupies is rational to the depth of the recursive calls, $O(h)$, h is the height of the recursive tree.

5.5.1 Summary

For your convenience, we prove a table that shows the frequent used recurrence equations' time complexity.

5.5.2 More Examples

5.1 Pow(x, n) (50).

Solution: $T(n) = T(n/2) + O(1)$, the complexity is the same as the binary search, $O(\log n)$.

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 5.4: The cheat sheet for time and space complexity with recurrence function. If $T(n) = T(n-1)+T(n-2)+...+T(1)+O(n-1) = 3^n$

```

1 def myPow( self , x , n):
2     """
3         :type x: float
4         :type n: int
5         :rtype: float
6     """
7     if n==0:
8         return 1
9     if n<0:
10        n=-n
11        x=1.0/x
12    def helper(n):
13        if n==1:
14            return x
15
16        h = n//2
17        r = n-h
18        value = helper(h) #T(n/2) , then we have O(1)
19        if r==h:
20            return value*value
21        else: #r is going to be 1 bigger
22            return value*value*x
23    return helper(n)

```

5.6 Time VS Space Complexity

Normally the analysis of time complexity is way more difficult and complex compared with the analysis of space.

Trade space for time efficiency or trade time for space efficiency: Either running time or the physical space is more important to the algorithms depends on the real case. For example, if you put your algorithm on a backend server, we need to response the request of users, then decrease the response time is 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.

Therefore, to complete the story, for input with data size n , if we can get time complexity $O(n)$, which is called liner time. This is a very good performance. If we do better, in some cases, you can get $O(1)$, which means constant complexity, however, we only get constant complexity in either time or space, not both. There are complexity denoted as $O(n^d)$, we call power order, and for $O(d^n)$, we call this exponentially complexity. There are also the case of $O(\lg n)$, logarithm of n , which sometimes come together with n as $O(n \lg n)$. By seeing more complexity algorithms in Part III, ?? and ?? we can have more sense of the complexity analysis.

5.6.1 Big-O Cheat Sheet

In this section, we provide the plotting of common seen time complexity functions (shown in Fig 5.2): including $\log_2 n$, n , $n \log_2 n$, n^2 , 2^n , and $n!$, so that we can sense the complexity change as the input size n increase. Resource found on <http://bigocheatsheet.com/>.

Table 5.1: Explanation of Common Growth Rate

Growth Rate	Name	Example operations
$O(1)$	Constant	append, get item, set item
$O(\log n)$	Logarithmic	binary search in the sorted array
$O(n)$	Liner	Copy, iteration
$O(n \log n)$	Linear-Logarithmic	MergeSort, QuickSort
$O(n^2)$	Quadratic	Nested Loops
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	Backtracking, Combination
$O(n!)$	factorial	Permutation

Also, we provide the average and worst time and space complexity for the some classical data structure's operations (shown in Fig. 5.5) and of

algorithms (shown in Fig. 5.5).

Data Structure	Common Data Structure Operations									Space Complexity Worst	
	Time Complexity				Worst						
	Average	Search	Insertion	Deletion	Access	Search	Insertion	Deletion			
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log(n))$	$\Theta(n)$	
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	
AVL Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	

Algorithm	Time Complexity				Space Complexity Worst	
	Best		Average			
	Worst	Worst	Worst	Worst		
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n \log(n))$	$O(\log(n))$	
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$	
Timsort	$\Theta(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$	
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$	
Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	
Tree Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n^2)$	$O(n)$	
Shell Sort	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$	
Bucket Sort	$\Theta(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n^2)$	$O(n)$	
Radix Sort	$\Theta(nk)$	$\Theta(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$	
Counting Sort	$\Theta(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$	
Cubesort	$\Theta(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$	

Figure 5.5: Complexity of Common Data structures

5.7 Exercises

5.7.1 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. $T(n) = T(n - 1) + T(n - 2) + T(n - 3) + \dots + T(1) + O(1)$.

Part III

Footstone: Bit Manipulation and Data Structures

There seems no book that put bit manipulation at the very beginning of a data structure and algorithm book. However, it plays big role for us to understand how different data type is stored and manipulated in the closest and efficient way a non-hardware programming language can achieve. Knowing how to manipulate bits and know its applications can also help us device more efficient algorithms in the later Chapter.

In the programming, data structures are used to store data and make operations on them so that we can conduct different algorithms on them in order to solve real-world problems and meet certain efficiency. The comparison between varies of data structures are highly dependable on the context of the problem we are facing. Being familiar with data structures is a must for us to understand and implement algorithms following this part. The concepts of data structures and the real data types and/or built-in modules in Python goes hand in hand for the real understanding. Thus, in our book, we insist on learning the concepts, real implementation with basic built-in data types: list / dict / string / together. On this base, we learn built-in modules which implements these data structures for us directly and with good efficiency.

On high level, data structures can be categories as two main types: *Liner* (Chapter 7) and *Non-liner* (include: Heap and Graph in Chapter ?? and Chapter ??). Before we move ahead to learn these data structures, it is essential for us to understand how normally data structures are categorized based on specific characters:

- **Mutable vs Immutable** In the sense of if modification of the items of the data structures is allowed, there are *mutable* and *immutable* data structures.
- **Static vs Dynamic** Moreover, we can categorize the data structures as *static data structures* and *dynamic data structures* according to if we can change the size of the created data structures. In static data structure the size of the structure is fixed since its creation. While, in dynamic data structure, the size of the structure is not fixed and can be modified through operations such as Insertion and Append. Dynamic data structures are designed to facilitate change of data structures in the run time.

The implementation of different data structures can vary as the programming languages. To make the contents more compact and make the reference more convenient, in this part, we combine data structures from the programming literature with corresponding data structures (either built-in or external modules) come from Python. Due to this understanding, for each data structure, the contents are organized as:

- firstly we will introduce the concept of the data structures including definition, pros, and cons;

- secondly, the common basic operations with concepts and time complexity: Access, Search, Insertion, Deletion.
- lastly, to complete the picture, we introduce Python’s data structures (either built-in or external) with their methods and corresponding operations.

Divide and Conquer serves as the fundamental problem solving methodology for the software programming, Data structures on the other hand plays the role of laying the foundation for any problem-solving paradigm or say algorithms to run on. Therefore, the content of this chapter will serve as the footstone for the purpose of the whole book – “crackin” the LeetCode problems. The purpose of this part is to give beginners a chance to learn different data structures and its Python implementation systematically and practically in the sense of problem solving. For medium or higher level audiences, the organization of this part can help them review their knowledge base efficiently.

6

Bit Manipulation

In this chapter, we will introduce basic knowledge about bit manipulation using Python. We found that a lot of popular Python book or even algorithm book they do not usually cover the topic of bit manipulation. However, mastering some basics operators, properties and knowing how bit manipulation sometimes can be applied to either save space or time complexity of your algorithm.

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.

6.1 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

```

6.2 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 6.1: 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

```

6.3 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.

$$\sum_{i=0}^{N-1} 2^i = 2^{(N-1)} + 2^{(N-2)} + \dots + 2^2 + 2^1 + 2^0 = 2^N - 1 \quad (6.1)$$

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. 6.1.

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:

$$5_{10} = 0000,0101_2, \quad (6.2)$$

$$-5_{10} = 1111,1010_2 + 1_2, \quad (6.3)$$

$$-5_{10} = 1111,1011_2 \quad (6.4)$$

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 two_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 two_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.

6.4 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. 6.2 and 6.4, 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

```

6.5 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

6.1 136. 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 6.1.

```
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
```

6.2 137. 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

```

6.3 421. 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

6.4 190. 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
```

6.5 201. 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
```

6.6 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.

7

Linear Data Structure

The focus of this chapter includes:

- Understanding the **concept of Array** data structure and its **basic operations**;
- Introducing the built-in data structures include **list, string, and tuple** which are arrays but each come with different features and popular module as a complement;
- Understanding the concept of **the linked list**, either single linked list or the doubly linked list, and the Python implementation of each data structure.

7.1 Array

An 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. 7.1. 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 other items in the array with $O(1)$. 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.

Static Array VS Dynamic Array There are two types of array: static array and dynamic array. They are different in the matter of fixing size or

not. In the static array, once we declared the size of the array, we are not allowed to insert or delete any item at any position of the array. This is due to the inefficiency of doing so, which can lead to $O(n)$ time and space complexity. For dynamic array, the fixed size restriction is removed but with high price to allow it to be dynamic. However, the flip side of the coin is that 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.

Commonly, arrays are used to implement mathematical vectors and matrices. Also, arrays are the basic units implementing other data structures, such as hashtables, heaps, queues, stacks. 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. *To note that it is not necessarily for array data structure to have the same data type in the real implementation of responding programming language as Python.*

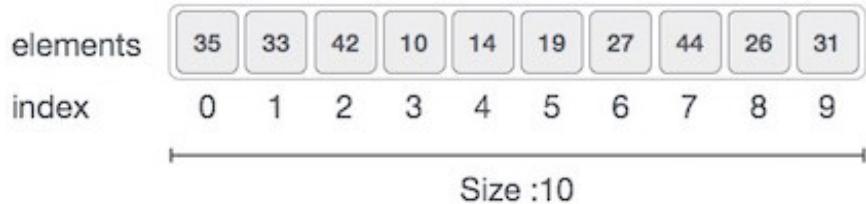


Figure 7.1: Array Representation

Operations Array supports the following operations:

- 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 array due to the fact that we need to shift the items after the modified position;
- 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.

In Python, there is no strictly defined built-in data types that are static array. But it does have three there are built-in Array-like data structures: List, Tuple, String, and Range. These data structures are different in the sense of mutability, static or dynamic. More details of these data structures in Python will be given in the next section. **module array** which is of same

data types just as the definition of Array here is also implemented the same as in C++ with its array data structure. However, **array** is not as widely used as of these three Items in list are actually not consecutive in memory because it is mutable object. Memory speaking, list is not as efficient as module Array and Strings.

7.1.1 Python Built-in Sequence: List, Tuple, String, and Range

In Python, *sequences* are defined as ordered sets of objects indexed by non-negative integers. *Lists* and *tuples* are sequences of arbitrary objects. While *strings* are sequences of characters. Unlike List, which is mutable¹ and dynamic², strings and tuples are immutable. All these sequence type data structures share the most common methods and operations shown in Table 7.1 and 7.2. To note that in Python, the indexing starts from 0.

Table 7.1: Common Methods for Sequence Data Type in Python

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

Table 7.2: Common Methods for Sequence Data Type in Python

Operation	Description
<code>s + r</code>	Concatenates two sequences of the same type
<code>s * n</code>	Make n copies of s, where n is an integer
<code>v₁, v₂, ..., v_n = s</code>	Unpack n variables from s
<code>s[i]</code>	Indexing-returns i th element of s
<code>s[i:j:stride]</code>	Slicing-returns elements between i and j with optimal stride
<code>x in s</code>	Return <i>True</i> if element x is in s
<code>x not in s</code>	Return <i>True</i> if element x is not in s

¹can modify item after its creation
²

Negative indexing and slicing For indexing and slicing, we can pass by negative integer as index and stride. The negative means backward, such as -1 refers to the last item, -2 to the second last item and so on. For the slicing

We list other characters and operations for each of these sequence data types:

List and Range

A list is similar to an array, but has a variable size which makes it more like a dynamic array, and does not necessarily need to be made up of a single continuous chunk of memory. While this does make lists more useful in general than arrays, you do incur a slight performance penalty due to the overhead needed to have those nicer characteristics. Also, list can be composed of items of any data types: including boolean, int, float, string, tuple, dictionary or even list itself. A list can be easily embedded into a list, which makes multi-dimensional data structure. A list is an ordered collection of items just like the definition of array.

A list is a *dynamic mutable* type and this means you can add and delete elements from the list at any time. `list` is optimized for fast fixed-length operations as the definition of Array. It is dynamic, thus it supports size growth, however, it will incur $O(n)$ memory movement costs for `pop(0)` and `insert(0, v)` operations which change both the size and position of the underlying data representation.

The `range()` type returns an immutable sequence of numbers between the given start integer to the stop integer. `range()` constructor has two forms of definition: `range(stop)` and `range([start], stop[, step])`: it is used to generate integers in range $[start/0, stop]$, and the step can be positive/negative integer which determines the increment between each integer in the sequence.

Use List as Static Array Therefore, list is better used with fixed size, and no operation that incur items shifting such as `pop(0)` and `insert(0, v)`, or operation that incurs size growth such as `append()`. Because, it pre-alloc a fixed size and once the size larger than this size, a new larger array is made and everything inside the old array is copied over, then the old array is marked for deletion. For example, we new a fixed size list, and we can do slicing, looping, indexing.

```

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 element.

```

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

```

Use List as Dynamic Array When the input size is reasonable, list can be used dynamically. Now, Table 7.3 shows us the common List Methods, and they will be used as list.methodName().

Table 7.3: Common Methods of List

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

Now, let us look at some exemplary code.

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]

```

INSERT and APPEND: To insert an item into the list, it actually involves one position shift to all the items that are after this position. This makes it takes $O(n)$ time complexity.

```

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]
```

The time complexity of different built-in method for list can be found at wiki.python.org/moin/TimeComplexity.

If you have a lot of numeric arrays you want to work with then it is worth using the **NumPy** library which is an extensive array handling library often used by software engineers to do linear algebra related tasks.

String

String are similar to static array and it follows the restriction that it only stores one type of data: characters represented using ASCII or Unicode³. String is more compact compared with storing the characters in *list*. In all, string is immutable and static, meaning we can not modify its elements or extend its size once its created.

String is one of the most fundamental built-in data types, this makes managing its common methods shown in Table 7.4 and 7.5 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.

Following this, we give some examples showing how to use these functions.

join(), split(), and replace() The str.join(), str.split(), and str.replace() methods are a few additional ways to manipulate strings in Python.

³In 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.

Table 7.4: 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 7.5: 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

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 #Ouput
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 #Ouput
3 abc

```

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.

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.

Related Useful Functions 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))
```

Tuple

A tuple is a sequence of immutable Python objects, which is to say the values of tuples can not be changed once its assigned. Also, as an immutable objects, they are hashable, and thus be used as keys to dictionaries. Like string and lists, tuple indices start at 0, and they can be indexed, sliced, concatenated and so on. Tuples only offer two additional methods shown in Table 7.6.

Since, tuples are quite similiar to lists, both of them are used in similar situations as well. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

Table 7.6: Methods of Tuple

Method	Description
count(x)	Return the number of items that is equal to x
index(x)	Return index of first item that is equal to x

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

New and Initialize Tuple Tuple can be created in two different syntax: (1) putting different comma-separated values in a pair of parentheses; (2) creating a tuple using built-in function *tuple()*, if the argument to tuple() is a sequence then this creates a tuple of elements of that sequences. See the Python snippet:

```

1  ''' new a tuple '''
2
3 # creat with ()
4 tup = () # creates an empty tuple
5 tup1 = ('crack', 'leetcode', 2018, 2019)
6 tup2 = ('crack', ) # when only has one element , put comma behind
7     , so that it wont be translated as string
8
9 # creat with tuple()
10 tup3 = tuple() # new an empty tuple
11 tup4 = tuple("leetcode") # the sequence is passed as a tuple of
12     elements
13 tup5 = tuple(['crack', 'leetcode', 2018, 2019]) # same as tuple1
14 print('tup1: ', tup1, '\ntup2: ', tup2, '\ntup3: ', tup3, '\n
15     tup4: ', tup4, '\ntup5: ', tup5)

```

The out put is:

```

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 A tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed. We can also assign a tuple to different values (reassignment).

```

1 '''change a tuple'''
2 tup = ('a', 'b', [1, 2, 3])
3 #tup[0] = 'c' #TypeError: 'tuple' object does not support item
               assignment
4 tup[-1][0] = 4
5 print(tup)
6 tup = ('c', 'd')
7 print(tup)
```

The output is:

```

1 ('a', 'b', [4, 2, 3])
2 ('c', 'd')
```

Deleting a Tuple As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple. But deleting a tuple entirely is possible using the keyword del.

```

1 del tup
2 print(tup)
```

After del, when try to use tup again it returns NameError.

```
1 NameError: name 'tup' is not defined
```

7.1.2 Bonus

Circular Array The corresponding problems include:

1. 503. Next Greater Element II

7.1.3 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.

```

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]
8
9         if type.isnumeric():
10             digits.append(log)
11         else:
12             letters.append((splited[1:], id))
13     letters.sort() #default sorting by the first element
                    and then the second in the tuple
14
15     return [id + ' ' + other for other, id in letters] +
            digits

```

```

1 def reorderLogFiles(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
```

7.2 Linked List

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers. The benefits of linked lists include: first, they do not require sequential spaces; second, they can start small and grow arbitrarily as we add more items to the data structures. Linked list is designed to offer flexible change of size and constant time complexity for inserting a new element which can not be obtained from array.

Even in Python, lists are actually dynamic arrays, However, it still requires growing by copy and paste periodically. However, linked list suffers from its own demerits:

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2. Extra memory space for a pointer is required with each element of the list.

The composing unit of linked list is called **nodes**. There are two types of linked lists based on its ability to iterate items in different directions: Singly Linked List wherein a node has only one pointer to link the successive node, and Doubly Linked List wherein a node has one extra pointer to link back to its predecessor.

We will detail these two sub data structures of linked list in the following sections.

7.2.1 Singly Linked List

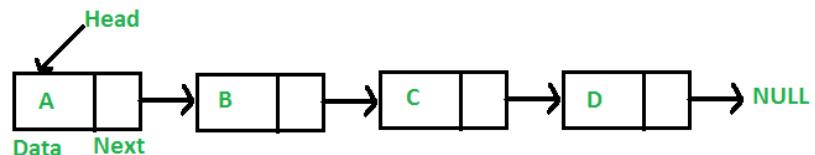


Figure 7.2: Linked List Structure

Fig. 7.2 shows the structure of a singly linked list. As we can see, a singly linked is a linear data structure with only one pointer between two successive nodes, and can only be traversed in a single direction, that is, we

can go from the first node to the last node, but can not do it in backforward direction.

Node To implement a singly linked list, we need to first implement a Node which has two members: **val** which is used to save contents and **next** which is a pointer to the successive node. The Node class is given as:

```

1 class SinglyListNode(object):
2     def __init__(self, val = None):
3         self.val = val
4         self.next = None

```

Implementation Here, we define a class as **SinglyLinkedList** which implements the operations needed for a singly linked list data structure and hide the concept of Node to users. The **SinglyLinkedList** usually needs a head that points to the first node in the list, and we need to make sure the last element will be linked to a **None** node. The necessary operations of a linked list include: insertion/append, delete, search, clear. 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

```

1 class SinglyLinkedList:
2     def __init__(self):
3         # with only head
4         self.head = None
5         self.size = 0
6     def len(self):
7         return self.size

```

Append: it is a common scenario that we build up a linked list from a list, which requires append operations. Because in our implementation, the head pointer always points to the first node, it requires us to traverse all the nodes to implement the Append, which gives $O(n)$ as the time complexity for append.

```

1 #...
2 def append(self, val):
3     node = SinglyListNode(val)
4     if self.head:
5         # traverse to the end
6         current = self.head
7         while current:
8             current = current.next
9             current.next = node
10    else:
11        self.head = node
12        self.size += 1

```

Deletion: sometimes we need to delete a node by value in the linked list, this requires us to rewire the pointers between the predecessor and successor of

the deleting node. This requires us to find iterate the list to locate the node to be deleted and track the previous node for rewiring. We can possibly have two cases:

1. if the node is head, directly repoint the head to the next node
2. otherwise, we need to connect the previous node to current node's next node, and the head pointer remains untouched.

```

1 #...
2     def delete(self, val):
3         current = self.head
4         prev = self.head
5         while current:
6             if current.val == val:
7                 # if the node is head
8                 if current == self.head:
9                     self.head = current.next
10                # rewire
11                else:
12                    prev.next = current.next
13                    self.size -= 1
14                    prev = current
15                    current = current.next

```

Sometimes, we will be asked to delete a List node, this deleting process does not need the head and the traversal to find the value. We simply need to change the value of this node to the value of the next node, and connect this node to the next node's node

```

1 def deleteByNode(self, node):
2     node.val = node.next.val
3     node.next = node.next.next

```

Search and iteration: in order to traverse the list and not to expose the users to the node class by usig node.val to get the contents of the node, we need to implement a method **iter()** that returns a generator gives out the contents of the list.

```

1 # ...
2     def iter(self):
3         current = self.head
4         while current:
5             val = current.val
6             current = current.next
7             yield val

```

Now, the linked list iteration looks just like a normal like iteration. Search operation can now built upon the iteration and the process is the same as linear search:

```

1 # ...
2     def search(self, val):

```

```

3     for value in self.iter():
4         if value == val:
5             return True
6     return False

```

Clear: in some cases, we need to clear all the nodes of the list, this is a quite simple process. All we need to do is to set the head to None

```

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

```

7.2.2 Doubly Linked List

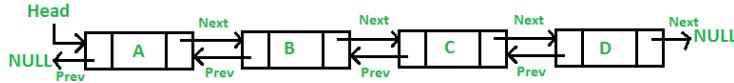


Figure 7.3: Doubly Linked List

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. Because of the prev pointer, a DLL can traverse in both forward and backward direction. Also, compared with SLL, some operations such as deletion is more efficient because we do not need to track the previous node in the traversal process.

```

1 # Node of a doubly linked list
2 class Node:
3     def __init__(self, val, prev = None, next = None):
4         self.val = val
5         self.prev = prev # reference to previous node in DLL
6         self.next = next # reference to next node in DLL

```

We define our class as DoublyLinkedList. Same as class SinglyLinkedList, have one pointer called **head** and another variable to track the size. Therefore we skip the definition of the class and its init function.

Append: The only difference is to link the nodes when adding and relinking when deleting.

```

1 # linking
2 # replace line 3, line 9
3 node = Node(val)
4 current.next, node.prev = node, current

```

Deletion: compared with sll, we do not need to track the previous node.

```

1 #...
2     def delete(self, val):
3         current = self.head
4         #prev = self.head
5         while current:
6             if current.val == val:
7                 # if the node is head
8                 if current == self.head:
9                     current.prev = None #set the prev
10                    self.head = current.next
11                    # rewire
12                else:
13                    #prev.next = current.next
14                    current.prev.next, current.next.prev =
15                    current.next, current.prev
16
17                    self.size -= 1
18                    #prev = current
19                    current = current.next

```

All the remaining operations such as Search, Iteration, and Clear are exactly the same as in sll.

7.2.3 Bonus

Tail Pointer For both singly and doubly linked list, if we add another **tail** pointer to its class, which points at the last node in the list, can simplify some operations of the linked list from $O(n)$ to $O(1)$.

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 that 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. This has the same usage of adding the tail pointer mentioned before. While, the other side of the flip coin is when we are iterating the nodes, we need to compare current visiting node with the head node and when we make sure we end the iteration after visiting the tail node when the next points to the head pointer.

And for circular linked list, to iterate all items, we need to set up the end condition for the while loop. If we let the current node start from the head node (the head is not None), the loop will terminate if the next node is head node again.

```

1 def iterateCircularList(head):
2     if not head:
3         return
4     cur = head
5     while cur.next != head:
6         cur = cur.next
7     return

```

Dummy Node Dummy node is a node that does not hold any value – an empty Node use None as value, but is in the list to provide an extra node at the front and/or read of the list. It is used as a way to reduce/remove special cases in coding so that we can simply the coding complexity.

Divide and Conquer + Recursion With the coding simplicity of recursion and the ability to iterate in a backward (or bottom-up) direction, we can use divide and conquer to solve problems from the smallest problem. The practical experience say that this method can be very helpful in solving linked list problems.

Let's look at a very simple example on LeetCode which demonstrates both the usage of dummy node and recursion.

7.1 83. Remove Duplicates from Sorted List (easy). 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 traverse the list and compare the current node's value with the next's to check its equivalency: (1) if YES: delete the next code and not move the current node; (2) if NO: we can move to the next node. We can also solve it in recursion way. We recursively call the node.next and the end case is when we meet the last node, we return that node directly. Then in the bottom-up process, we compare the current node and the returning node (functioning as a head for the subproblem).

Solution 1: Iteration. 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          # current pointer wont move unless the next has
13          # different value
14          if current.next and current.val == current.next
15              .val:
16              # delete next
17              current.next = current.next.next
18          else:
19              current = current.next
20      return head
21
22  return iterative(head)

```

We can see each time we need to check if `current.next` has value or not, this process can be avoid using a dummy node before the head.

```

1 # use of dummy node
2 def iterative(head):
3     dummy = ListNode(None)
4     dummy.next = head
5     current = dummy
6     while current.next:
7         # current pointer wont move unless the next has
8         # different value
9         if current.val == current.next.val:
10             # delete next
11             current.next = current.next.next
12         else:
13             current = current.next
14     return head

```

Solution 2: Recursion.

```

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         # delete next
8         node.next = node.next.next
9     return node

```

7.2.4 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)

7.3 Stack and Queue

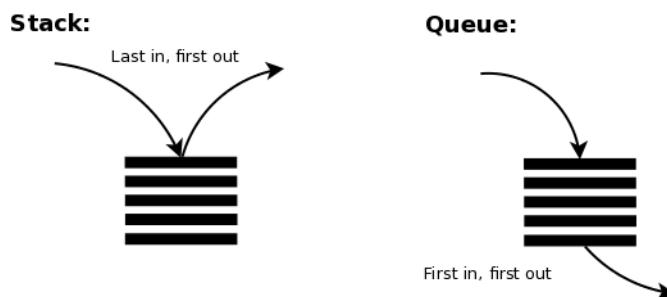


Figure 7.4: Stack VS Queue

Stacks and queue are dynamic arrays with restrictions on deleting elements. Stack data structure can be visualized as a stack of plates, we would always put a plate on top of the pile, and get one from the top of it too. This is stated as **Last in, first out (LIFO)**. Queue data structures are like real-life queue in the cashier out line, it follows the rule 'first come, first served', which can be officially as **first in, first out (FIFO)**.

Therefore, given a dynamic array, we always add element by appending at the end, a stack and a queue can be implemented with prespecified deleting operation: for stack, we delete from the rear; for a queue, we delete from the front of the array.

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 13, (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 13, (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.

7.3.1 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.

7.3.2 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 7.7.

Table 7.7: 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.

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()
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.

7.3.3 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 Table 7.8. *In python 3, we use lower case queue, but in Python 2.x it uses Queue, in our book, we learn Python 3.*

Table 7.8: 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 7.9.

Table 7.9: 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
```

7.3.4 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. 7.5. *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:

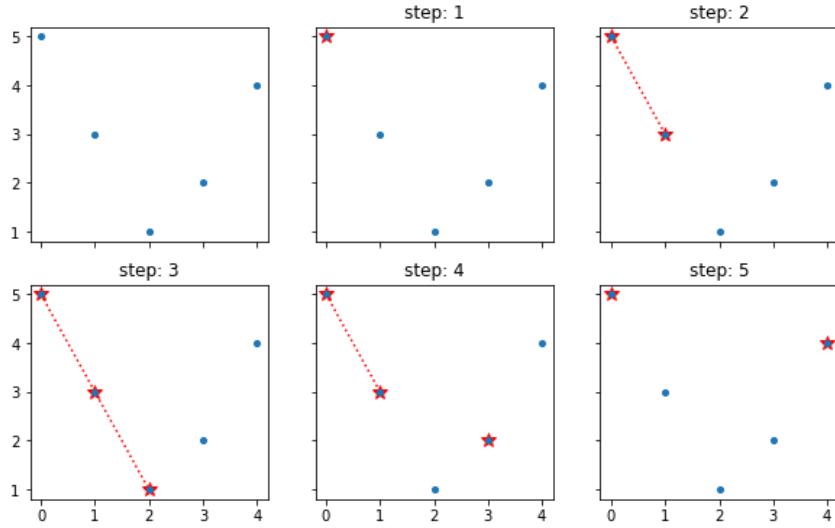


Figure 7.5: The process of decreasing monotone stack

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

```

7.2 907. 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))
16 ) % 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)

```

7.3.5 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.

7.3.6 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
   current ping.
8
9 It is guaranteed that every call to ping uses a strictly larger
   value of t than before.
10
11
12 Example 1:
13
14
15 Input: inputs = ["RecentCounter","ping","ping","ping","ping"],
           inputs = [[], [1], [100], [3001], [3002]]
16 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()
```

```

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
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
5. 42 Trapping Rain Water
6. 739. Daily Temperatures
7. 321. Create Maximum Number

7.4 Hash Table

A hash map (or hash table) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function $h(key)$ to compute an index into an array of buckets or slots, from which the desired value will be stored and found. A well-designed hashing should gives us constant average time to insert and

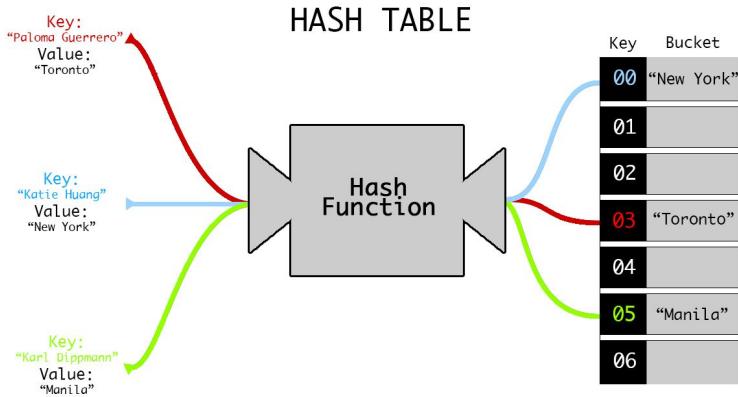


Figure 7.6: Example of Hashing Table

search for an element in a hash map as $O(1)$. In this section, we will examine the hashing design and analysis mechanism.

First, let us frame the hashing problem: given a universe U of keys (or items) with size n , with a hash table denoted by $T[0 \dots m - 1]$, in which each position, or slot, corresponds to a key in the hash table. Fig. 7.6 shows an example of hash table. When two keys have the same hash value produced by hash function, it is called **collision**. Because $\alpha > 1$, there must be at least two keys that have the same hash value; avoiding collisions altogether seems impossible. Therefore, 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.

Applications If average lookup times in an algorithm of each item is n and each time, using linear search will take $O(n)$, this makes the total time complexity to $O(n^2)$. If we spend $O(n)$ to save them in the hash table and each search will be $O(1)$, thus decrease the total time complexity to $O(n)$. For example, string process such as Rabin-Karp algorithm for pattern matching in a string in $O(n)$ time. Also, there are two data structures offered by all programming languages uses hashing table to implement: hash set and hash map, in Python, there are *set* and *dict* (dictionary) built-in types.

- Hash Map: hash map is a data structures that stores items as (key, value) pair. And “key” are hashed using hash table into an index to access the value.
- Hash Set: different to hash map, in a hash set, only keys are stored and it has no duplicate keys. Set usually represents the mathematical notion of a set, which is used to test membership, computing standard

operations on such as intersection, union, difference, and symmetric difference.

7.4.1 Hash Function Design

Hash function maps the universe U of keys into the slots of a hashtable $T[0..m - 1]$. With hash function the element is stored in $f(k, m)$. $h : U \rightarrow \{0, 1, \dots, m - 1\}$. The hash function design includes two steps: interpreting keys as nature numbers and design hashing functions.

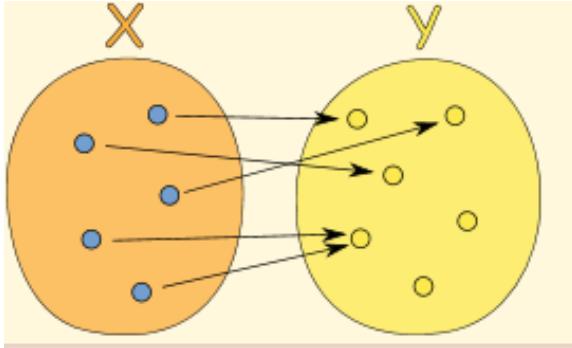


Figure 7.7: The Mapping Relation of Hash Function

Interpreting Keys For a given key, if the keys are not natural numbers, such as any string, or tuple, they need to be firstly interpreted as natural integers $N = \{0, 1, 2, \dots\}$. This interpretation relation(function) needs to be one to one; given two distinct keys, they should never be interpreted as the same natural number. And we denote it as a interpret function $k = f(key)$, where key is a input key and k is the interpreted natural number. For string or character, one possible way is to express them in suitable radix notation. we might translate “pt” as the pair of decimal integers (112, 116) with their ASCII character; then we express it as a radix128 integer, then the number we get is $(112 \times 128) + 116 = 14452$. This is usually called **polynomial rolling hashing**, to generalize, $k = s[0] + s[1] * p + s[2] * p^2 + \dots + s[n - 1] * p^{n-1}$, where the string has length n , and p is a chosen prime number which is roughly equal to the number of characters in the input alphabet. For instance, if the input is composed of only lowercase letters of English alphabet, $p=31$ is a good choice. If the input may contain both uppercase and lowercase letters, then $p=53$ is a possible choice.

Hash Function Design As the definition of function states: a function relates **each element** of a set with **exactly one element** of another set (possibly the same set). The hash function is denoted as $index = f(k, m)$. One essential rule for hashing is if two keys are equal, then a hash function

should produce the same key value ($f(s, m) = f(t, m)$, 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. The relation is denoted in Fig. 7.7. 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. There are generally four methods:

1. **The Direct addressing method**, $f(k, m) = k$, 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**, $f(k, m) = k \% m$, 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 number 4, then when we apply the division, it became $(key/c)/(m/c)$; both the quotient(also a multiple of the bucket size) and the remainder(modulo or bucket number) can be written as multiple of the common factor. So, the range of the slot index will be decrease to m/c . 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.

3. **The multiplication method**, $f(k, m) = \lfloor m(kA \% 1) \rfloor$. $A \in (0, 1)$ is a chosen constant and a suggestion to it is $A = (\sqrt{5} - 1)/2$. $kA \% 1$ means the fractional part of kA and equals to $kA - \lfloor kA \rfloor$. It is also shorten as $\{kA\}$. E.g. for 45.2 the fractional part of it is .2.
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 ch

7.4.2 Collision Resolution

Collision is unavoidable given that $m < n$ and the data can be adversary.

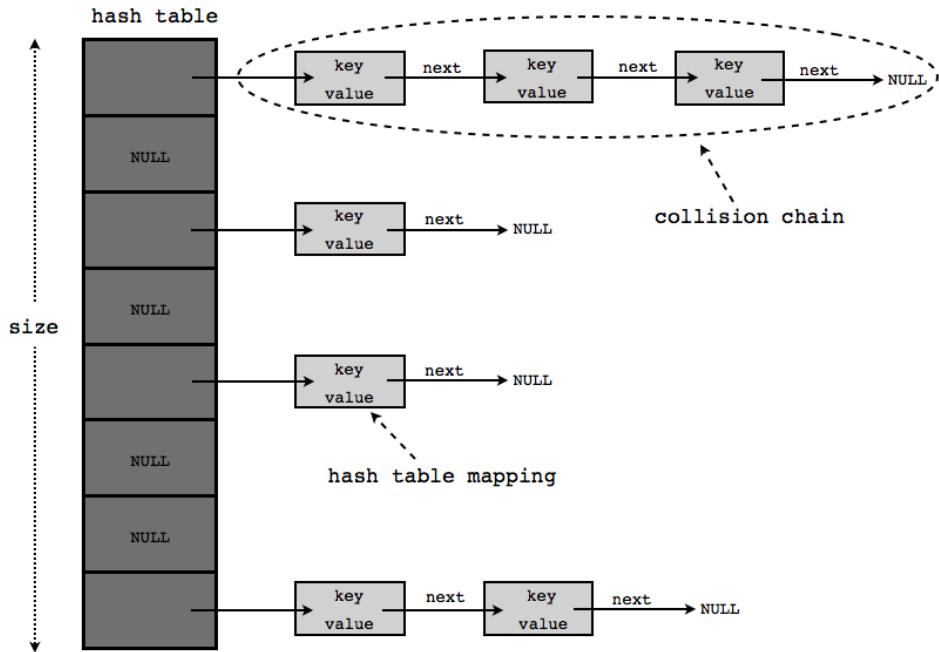


Figure 7.8: Hashtable chaining to resolve the collision

Resolving by 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 $f = k \% 4$, 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. 7.8.

The average-case time for searching and insertion is $O(\alpha)$ under the assumption of simple uniform hashing. However, the worst-case for operation can be $O(n)$ when all keys are mapped to the same slot.

The advantage of chaining is that hash table never fills up, and we can always add more elements by chaining behind. It is mostly used when the number of keys and the frequency of operations.

However, because using linked list, (1) the cache performance is poor; and (2) the use of pointers takes extra space that could be used to save data.

Resolving by Open Addressing Open addressing takes a different approach to handle the collisions, where all items are stored in the hash table. Therefore, open addressing requires the size of the hash table to be larger or equal to the size of keys ($m \geq n$). In open addressing, it computes a probe sequence as of $[h(k, 0), h(k, 1), \dots, h(k, m-1)]$ which is a permutation of $[0, 1, 2, \dots, m-1]$. We successively probe each slot until an empty slot is found.

Insertion and searching is easy to implement and are quite similar. How-

ever, when we are deleting a key, we can not simply delete the key and value and mark it as empty. If we did, we might be unable to retrieve any key during whose insertion we had probed slot i because we stop probing whenever empty slot is found.

Let us see with an example: Assume $\text{hash}(x) = \text{hash}(y) = \text{hash}(z) = i$. And assume x was inserted first, then y and then z . In open addressing: $\text{table}[i] = x$, $\text{table}[i+1] = y$, $\text{table}[i+2] = z$. Now, assume you want to delete x , and set it back to `NULL`. When later you will search for z , you will find that $\text{hash}(z) = i$ and $\text{table}[i] = \text{NULL}$, and you will return a wrong answer: z is not in the table.

To overcome this, you need to set $\text{table}[i]$ with a special marker indicating to the search function to keep looking at index $i+1$, because there might be element there which its hash is also i

Perfect Hashing

7.4.3 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 __init__(self):
4         self.table = [None] * 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, -1]:
19                self.slots[k] = (key, value)
20                return
21            i += 1
22        # double size and try again
23        self.slots = self.slots + [None]* self.size
24        self.size *= 2
25        return self.put(key, value)
26
27
28    def get(self, key: 'int') -> 'int':
29        """
30            Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key.
```

```

31     Returns the value to which the specified key is mapped,
32     or -1 if this map contains no mapping for the key
33     """
34     i = 0
35     while i < self.size:
36         k = self._h(key, i)
37         if not self.slots[k]:
38             return -1
39         elif self.slots[k][0] == key:
40             return self.slots[k][1]
41         else: # if its deleted keep probing
42             i += 1
43     return -1
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

```

7.4.4 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

7.4.5 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
Explanation: "testemail@leetcode.com" and "testemail@lee.tcode.
com" actually receive mails
25 Note:
26     1 <= emails[i].length <= 100
27     1 <= emails.length <= 100
28     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)
```

Trees

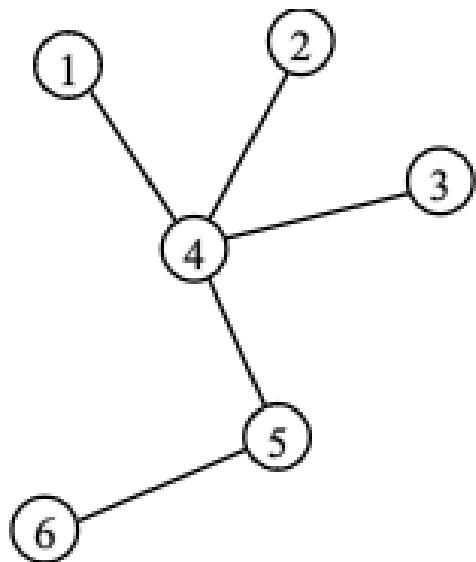


Figure 8.1: First shows a tree, and how it can be converted to an abstract tree data structure on the right side (put graph here).

Given an undirected graph with n vertices and $n - 1$ total edges as shown in Fig. 8.1 it is a *tree*. If we choose any node as a starting point, we relocate the nodes from this tree and we get our **rooted tree** structure, where the starting point of a tree is called a *root*. A **Polytree** is a directed acyclic graph (DAG) whose underlying undirected graph is a tree. The most commonly and generally referred tree data structure is essentially both a polytree and

a rooted tree.

However, trees are usually thought as a different data structure from graph. It is important to know why do we need to separate trees from the graph field? The reasons can be summarized as:

1. A tree is an easier data structure that can be recursively represented as a root node connected with its children.
2. 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.
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.

Due to the special condition of trees compared with graphs, things are easier. In this chapter, we firstly focus on:

1. Learning the terminologies of trees in Section 8.1.
2. Introducing binary tree which has at most two children only and its implementation in Section ??.
3. Stating the applications of trees by introducing different types of trees in the sense of how and what information it is storing and representing their data structures in Section ???. We introduce three types of search tree: Binary Search tree, Trie for string, Segment Tree.

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. At least 80% of 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.

How to learn Trees? Also, to be noticed, this chapter serves as an introduction, the summary and commonly seen problems related to tree will be detailed in Part ???. After the basics, we move on the core of tree algorithms, tree traversal and the method of divide and conquer in Chapter 15. More advanced tree algorithms will be found in Chapter ???.

8.1 Introduction and Terminologies

A tree is essentially a 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. A tree is defined as a collection of nodes, which are connected by edges.

Trees can be implemented as a graph with any of the graph representations we learned in the last chapter. A rooted tree is a tree in which a special node is singled out and be called **root** of the tree. Due to this, the generally defined trees are called **free tree**. **Forest** is a set of $n \geq 0$ disjoint trees.

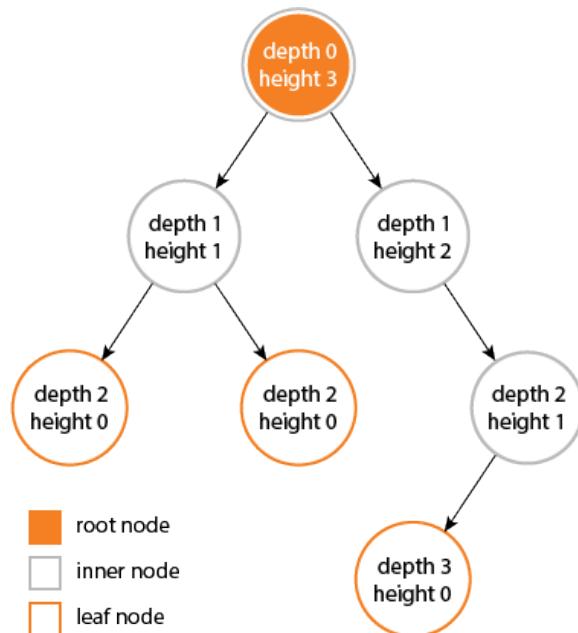


Figure 8.2: Example of a Tree with height and depth denoted

Rooted Trees

A rooted tree introduces a parent-child, sibling relationship between nodes. One example of the rooted tree is shown in Fig. 8.2. In general purposed programming and the coding interviews, a rooted tree is more widely used compared with the free tree. Thus, the rooted trees are one of the well-known non-linear data structures. They organize data hierarchically other than in the linear way.

The first node of the tree is called the **root**, if this root node is connected by another node, the root is then a parent node and the connected node is a child node instead. On the other hand, **leaves** are the last nodes on a tree.

They are nodes without children. Just like a real tree, we have the root, branches (subtree), and finally the leaves. Therefore, a rooted tree can be structured *recursively*: each node acts as the root of a subtree of the original tree. This subtree will contain the current root and all of its children.

Terminologies of Rooted Trees

Other than root, child, parent, siblings, subtree we mentioned before, there are more terminologies of nodes and trees used for the rooted trees.

Terminologies of Nodes

1. **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.
2. **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.
3. **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*.
4. **Ancestor:** The *ancestor* of a node is any node that is reachable by repeated proceeding from child to parent starting from this node.
5. **Degree:** The **degree** of a node is the number of its children. A leaf is necessarily degree zero.

Terminologies of Trees

1. **Height:** The *height*(or *depth*) of a tree would be the height of its root node, or equivalently, the depth of its deepest node.
2. **Diameter:** The *diameter* (or *width*) of a tree is the number of nodes (or edges) on the longest path between any two leaf nodes.
3. **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:
 - (a) Root->Leaf Path: the starting and ending node of the path is the root and leaf node respectively;
 - (b) Root->Any Path: the starting and ending node of the path is the root and any node (inner, leaf node) respectively;
 - (c) Any->Any Path: the starting and ending node of the path is both any node (Root, inner, leaf node) respectively.

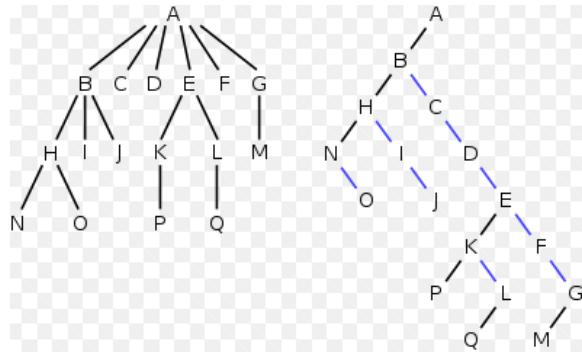


Figure 8.3: A 6-ary Tree Vs a binary tree.

8.2 N-ary Tree and Binary Tree

Among trees, if in a rooted tree, each node has no more than N children, it is called *N-ary Tree*. When $N = 2$, it is called *binary tree*. In binary tree, the two children are typically called *left child* and *right child*. In Fig. 8.3 shows a comparison of a 6-ary tree and a binary tree.

In this section, we focus on represent the trees with Python, and the next section, we talk about the applications of trees and show some example for N-ary tree and binary tree, plus implementation of the common operations used in trees: Insert, Delete, constructiton.

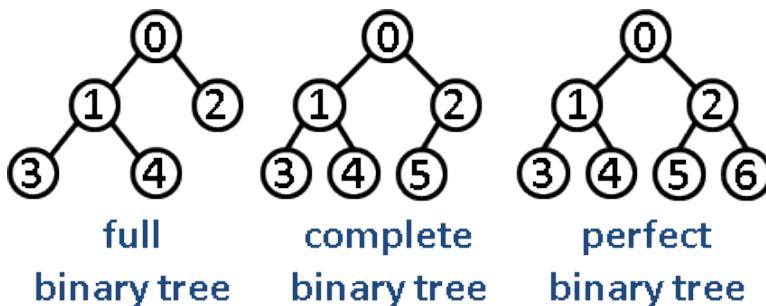


Figure 8.4: Example of different types of binary trees

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

1) Full Binary Tree, 2) Complete Binary Tree, 3) Perfect Binary Tree, 4) Balanced Binary Tree. And each we show one example in Fig. 8.4.

1. **Full Binary Tree:** A binary tree is full if every node has 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 and the number of all other non-leaf nodes has relation: $L = Non - L + 1$.

2. **Complete Binary Tree:** A Binary Tree is complete Binary Tree 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 Binary Tree 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 1.
5. **Degenerate (or pathological) tree:** A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

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 $2 * i + 2$.

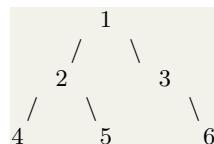
Define a Tree Node We define two classes each for the N-ary tree node and the binary tree node. A node is composed of a variable saving the data and children pointers to connect the nodes in the tree. The Python implementation of each is as:

```

1 class NaryNode:
2     '''Define a n-ary node'''
3     def __init__(self, n, val):
4         self.children = [None] * n
5         self.val = val
6
7 class BiaryNode:
8     '''Define a classical binary tree node'''
9     def __init__(self, val):
10        self.left = None
11        self.right = None
12        self.val = val

```

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 practice way is to treat the n-ary tree as a complete tree and be represented as an array first. For the above binary tree, because it is not complete, we pad the left tree of node 3 with None in the array, 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 trees of it will be $n * i + j, j \in [1, \dots, n]$. Thus, a better way to construct the above tree is from the array. We can automatize the tree construction recursively using the following code:

```

1 def constructTree(a, idx):
2     '''construct a binary tree recursively from input array a'''
3     if idx >= len(a):
4         return None
5     node = BinaryNode(a[idx])
6     node.left = constructTree(a, 2*idx + 1)
7     node.right = constructTree(a, 2*idx + 2)
8     return node

```

Now, we call this function, we will get the above tree:

```

1 nums = [1, 2, 3, 4, 5, None, 6]
2 root = constructTree(nums, 0)

```

There are some visualization code given in the code offered online to see the tree.

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.

8.3 LeetCode Problems

Graphs

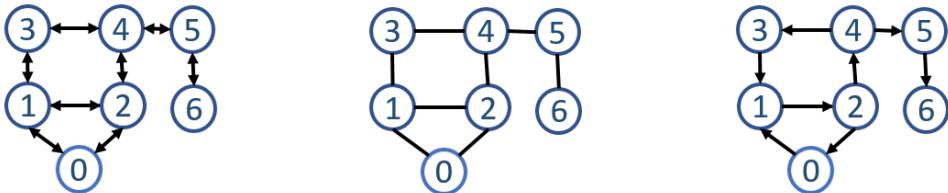


Figure 9.1: Example of undirected and directed graph: mark the vertices and edges

Graph is a natural way to represent connections and reasoning between things or events. A graph is made up of *vertices* (or nodes, points) are connected by *edges* (arcs, or lines). A graph structure is shown in Fig. 9.1. 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 in computer science.

In this chapter, 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.

As the first chapter related to graph, we mainly focus on explain related concepts, terminologies, graph representation, and the basic exhaustive search methods on real graph data structures:

- Knowing the definition and the **terminologies** commonly used in Sec-

tion 9.1.

- **Representing the graph** data structures and implement these representation in Python in Section 9.2.
- Learn the most basic graph search: Breath-first-search and Depth-first search in Section ??.

Arrangement of Graph in the Book Searching in graph which lies at the heart of the field of graph algorithms, therefore, we put effort in this book to explain the behavior, properties of them compared with a lot other books. More advanced searching techniques and applications will be detailed in Chapter 13 in the part IV. And more advanced graph algorithms that build upon the basic searching techniques will be taught in Chapter ?? in part VII. And graph related questions instead will be categorized in Chapter 30 in Part X.

Graph in Interviews Graph solution for some problems are most likely to be the naive solution, and it is a nice first step to give the naive algorithm design and analysis before moving on to more advanced solutions, such as divide and conquer, dynamic programming, or greedy algorithm. For some problems, graph search might be the only solution, so learning how to pruning the searching space with techniques like bidirectional search and backtracking would become handy.

9.1 Introduction and Terminologies

Graph is a widely used data structure to model real-world problems. A graph is a collection of *vertices* and *edges* (which connects two vertices). In Fig. 9.1, we have a graph with 7 vertices and 8 edges in the second graph. The *weights of edges* refers to the information with edges. 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.

Undirected Graph VS Directed Graph If all edges $e \in E$ is an unordered pair (u, v) , where $u, v \in V$ then the graph is undirected graph as shown in Fig. 9.1. On the other hand, if the edges are directed, for example, given an edge denoted as (u, v) which is ordered,in this book, we denote it as $(u \rightarrow v)$, meaning that the edge starts from u and point to v . This is as shown in Fig. 9.1. An edge in the undirected graph can be simply equal to two edges of (u, v) and (v, u) .

Embedded Graph VS Topological Graph A graph is defined as a set of vertices and a set of edges, these vertices do not have a geometric position of their own. Therefore, 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 the 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 then traversed, but built as we use them. The vertices of the implicit graph are the states of search vector in the backtrack we are going to learn in Chapter 9, while edges link pair of states that can be directly generated from each other. 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.

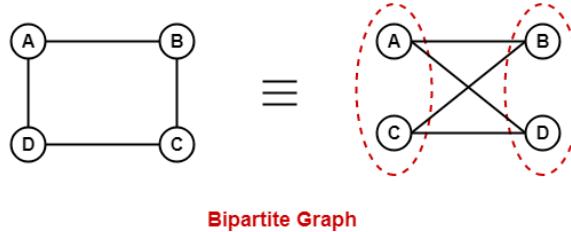


Figure 9.2: Bipartite Graph

Terminologies We can see more https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms.

1. **Path:** A path , a length of path in an unweighted graph is the number of edges it consists of. In the weighted graph, it may instead be the sum of the weights of edges that it uses. It is used to define the shortest/longest path and girth (shortest cycle length).
2. **Acyclic:** A graph is acyclic if it has no cycles. Acyclic directed graphs are often called directed acyclic graphs, and abbreviated as DAG.
3. **Tree:**
4. **Subgraph:** a subgraph of graph G is another graph formed from a subset of the vertices V_s and edges E_s of G . The V_s must include

all endpoints of the edge subset E_s , but may also include additional vertices. A **spanning subgraph** is one that includes all vertices of the graph V ; an **induced subgraph** is one that includes all the edges whose endpoints belong the vertex subset.

5. **Supergraph:** A graph formed by adding vertices, edges, or both to a given graph. If H is a subgraph of G , then G is a supergraph of H .
6. **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 graph vertices is denoted as $K_n = (n, 2) = n(n - 1)/2$.
7. **Bipartite:** A bipartite graph is a graph whose vertices can be divided into two disjoint sets V_1 and V_2 such that the vertices in each set are not connected to each other. 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. 9.2.
8. **Connected Graph:** A graph is connected each pair of vertices forms the endpoints of a path. The connected parts of a graph are called its components.

9.2 Graph Representation

Before move to the main contents, there are some Python 2-d array concepts and examples we need to cover first.

9.2.1 Python Two-dimensional Array

Two dimensional array is an array within an array. 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 data elements in two dimensional arrays 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 element in the inner array. If we mention only one index then the entire inner array 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 two-d array and intialize 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 array
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 m1 and m2, 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 the 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)]
```

 Try use numpy third party module?

```
1 import numpy as np
2 b = np.array(list) # convert list to array
3 col = b[:, 0] # access the first col
4 row = b[0, :] # access the first row
```

9.2.2 Four Types of Graph Representation

There are totally four ways to represent graph $G = (V, E)$ and store related edge information either it is directed or undirected: (1) Adjacency Matrix, (2) Adjacency List, (3) Edge List, and (4) Tree Structure. Each will be preferred to different situations. An example is shown in Fig 9.3.

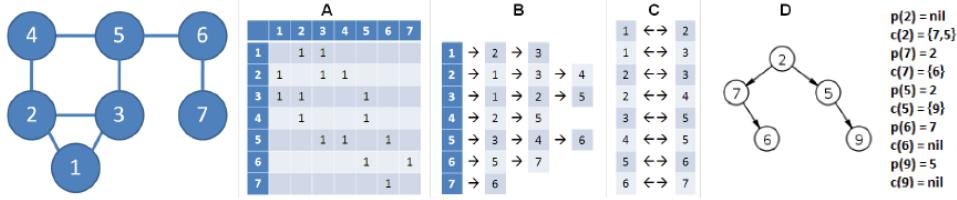


Figure 9.3: Four ways of graph representation, reenumerate it from 0.

For directed graph, the number of pairs of edges used in the data structure will be $|E|$. However, for the undirected graph, because one edge (u, v) denotes u and v are connected, so we can access v from u and it also works the other way around. Therefore, the number of edge pairs in the data structures will be doubled in the case of the total number of edges compared with the directed graph. Thus, it becomes $2|E|$.

Adjacency Matrix Adjacency matrices are in the form of a 2-D matrix and is the simplest type of graph representation. In a graph of $|V|$ nodes, the adjacency matrix will be $|V| \times |V|$. For undirected graph, it will be a symmetric matrix along the main diagonal as shown in A of Fig. 9.3. The matrix G is its own transpose: $G = G^T$. We can choose to store only the entries on and above the diagonal of the matrix, thereby cutting the memory needed in half. For unweighted graph, we can set edge from vertex u to v as $G[u][v] = 1$, and 0 otherwise. For a weighted graph it will be the $w(i, j)$ instead and 0 otherwise.

Adjacency matrix usually fits well to the dense graph where the edges are close to $|V|^2$, and thus only small ratio of the matrix will be blank. An adjacency matrix requires exactly $O(V)$ to enumerate the list of neighbors of a vertex v – an operation commonly used in many graph algorithms—even if vertex v only has a few of neighbors. For the above example, our matrix will be:

```
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
```

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, and for each vertex v we store another list of neighboring nodes with their vertex index as value. 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. We can use a normal 2-d array to represent the adjacent list. 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 takes $O(|V|)$ time complexity to do a simple linear search in the u -th row of the 2-d array. For the above example:

```
al = [[] for _ in range(7)]

# set 8 edges
al[0] = [1, 2]
al[1] = [2, 3]
al[2] = [0, 4]
al[3] = [1, 4]
al[4] = [2, 3, 5]
al[5] = [4, 6]
al[6] = [5]
```

If the graph is static, we can set the second dimension as a set or a dictionary to enable $O(1)$ search of an edge just as of in the adjacency matrix.

Edge List The edge list is a list of edges (one-dimensional), where each edge is possibly a tuple element. This type of representation helps us ordering the edges that serves the main graph algorithm as of in Kruskal's algorithm for Minimum Spanning Tree(MST) where the collection of edges are sorted by their length from shortest to longest. The edge list of the above figure will be:

```
el = []
el.extend(([0, 1], [1, 0]))
el.extend(([0, 2], [2, 0]))
el.extend(([1, 2], [2, 1]))
el.extend(([1, 3], [3, 1]))
el.extend(([3, 4], [4, 3]))
el.extend(([2, 4], [4, 2]))
el.extend(([4, 5], [5, 4]))
el.extend(([5, 6], [6, 5]))
```

Tree If the connected graph has no cycle and the edges $E = V - 1$, then there exists another form of representation with tree structure which we will learn more in the next section.

Using Dictionary Data Structure Except for the basic list of list, if you need a “name” for each node, or if we do not know the number of nodes, we can use Python dictionary instead. For the same 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. We directly convert the edge list into this format.

```

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:

```
{'a': {'b', 'c'}, 'b': {'a', 'd', 'c'}, 'c': {'e', 'b', 'a'}, 'd': {'e', 'b'}, 'e': {'f', 'c', 'd'}, 'f': {'e', 'g'}, 'g': {'f'}}
```

If we need weights for each edge, we can use two-dimensional dictionary. We the above graph, assume we assign $w(u, v) = u + v$.

```

1 dw = defaultdict(dict)
2 for v1, v2 in el:
3     dw[v1][v2] = v1 + v2
4 print(dw)

```

The output will be:

```
{0: {1: 1, 2: 2}, 1: {0: 1, 2: 3, 3: 4}, 2: {0: 2, 1: 3, 4: 6},
3: {1: 4, 4: 7}, 4: {3: 7, 2: 6, 5: 9}, 5: {4: 9, 6: 11}, 6:
{5: 11}}
```

9.3 Exercises

9.3.1 Knowledge Check

1. How to check if a graph is connected? A: We can check if a graph is connected by starting at an arbitrary node and finding out if we can reach all other nodes. (Both DFS and BFS works)
2. How to find cycle in a graph? A: A graph contains a cycle if during a graph search, we find a node whose neighbor has already been visited that marked as gray.

3. How to check if a graph is bipartite? A: A graph is bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with the same color. It is surprisingly easy to check if a graph is bipartite using graph traversal algorithms. The idea is to color the starting node blue, all its neighbors red, all their neighbors blue, and so on. If at some point of the search we notice that two adjacent nodes have the same color, this means that the graph is not bipartite. Otherwise the graph is bipartite and one coloring has been found. Note that in the general case, it is difficult to find out if the nodes in a graph can be colored using k colors so that no adjacent nodes have the same color. Even when $k \geq 3$, no efficient algorithm is known but the problem is NP-hard.

9.3.2 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)

10

Heap and Priority Queue

In this chapter, we introduce heap data structures which is essentially an array object but it can be viewed as a nearly complete binary tree. The concept of the data structures in this chapter is between liner and non-linear, that is using linear data structures to mimic the non-linear data structures and its behavior for higher efficiency under certain context.

10.1 Heap

Heap is a tree based data structures that satisfies **heap property** but implemented as an array data structure. There are two kinds of heaps: **max-heaps** and **min-heaps**. In both kinds, the values in the nodes satisfy a **heap property**. For max-heap, the property states as for each node in the heap at i , $A[p[i]] \leq A[i]$. Normally, heap is based on binary tree, which makes it a binary heap. Fig. 10.1 show a binary max-heap and how it looks like in a binary tree data structure. In the following content, we default our heap is a binary heap. Thus, the largest element in a max-heap is stored at the root. For a heap of n elements the height is $\log n$.

As we can see we can implement heap as an array due to the fact that the tree is complete. A complete binary tree is one in which each level must be fully filled before starting to fill the next level. Array-based heap is more space efficient compared with tree based due to the non-existence of the child pointers for each node. To make the math easy, we iterate node in the tree starting from root in the order of level by level and from left to right with beginning index as 1 (shown in Fig. 10.1). According to such assigning rule, the node in the tree is mapped and saved in the array by the assigned index (shown in Fig. 10.1). In heap, we can traverse the imaginary binary

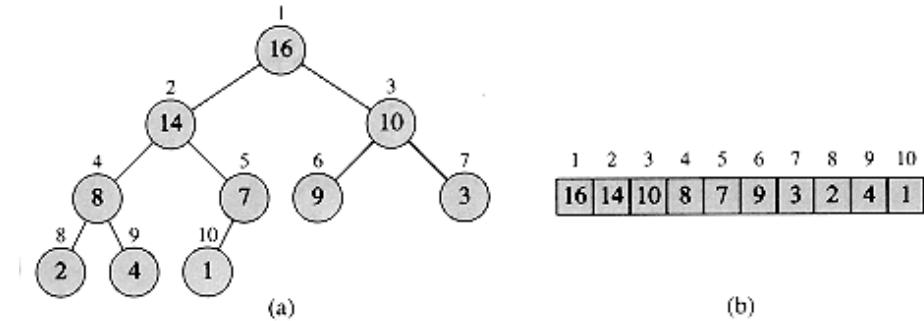


Figure 10.1: Max-heap be visualized with binary tree structure on the left, and implemnted with Array on the right.

tree in two directions: **root-to-leaf** and **leaf-to-root**. Given a parent node with p as index, the left child of can be found in position $2p$ in the array. Similarly, the right child of the parent is at position $2p+1$ in the list. To find the parent of any node in the tree, we can simply use $\lfloor p/2 \rfloor$. In Python3, use integer division $n//2$. Note: we can start index with 0 as used in `heapq` library introduced later in this section. Given a node x , the left and right child will be $2*x+1$, $2*x+2$, and the parent node will have index $(x-1)//2$.

The common application of heap data structure include:

- Implementing a priority-queue data structure which will be detailed in the next section so that insertion and deletion can be implemented in $O(\log n)$; Priority Queue is an important component in algorithms like Kruskal's for minimum spanning tree (MST) problem and Dijkstra's for single-source shortest paths (SSSP) problem.
- Implementing heapsort algorithm,

Normally, there is usually no notion of 'search' in heap, but only insertion and deletion, which can be done by traversing a $O(\log n)$ leaf-to-root or root-to-leaf path.

10.1.1 Basic Implementation

The basic methods of a heap class should include: **pop**, **push**, and **heapify**. **push** an item into the heap and **pop** the root item at the heap out, and still maintain the heap property. And **heapify** denotes the operation needed for an given array, to convert it to a heap directly and efficiently.

Let's implement a heap class using list. Because the first element of the heap is actually empty, we define our class as follows:

```

1 class Heap:
2     def __init__(self):
3         self.heap = [None]
```

```

4     self.size = 0
5     def __str__(self):
6         out = ''
7         for i in range(1, self.size + 1):
8             out += str(self.heap[i]) + '\n'
9         return out

```

Assuming we already have got a heap shown in Fig. 10.1, push or pop an item from the current heap requires us to do post-processing in order to maintain the heap property. Let's discuss the two cases. Change it to use max heap as example.

Push with Floating When we push an item into, to maintain the complete binary tree property, the new item goes to the end of the heap(array) first. Assuming the new item $a[i]$ is the smallest item up till now, there will be violation of the heap property through the $a[i] \rightarrow \text{root}$ path. To correct the potential violation, we traverse the path $a[i] \rightarrow \text{root}$, and compare each node and its parent to decide if a swap operation is needed. For a min-heap, if the child node is smaller than the parent, that is a violation, and we swap these two nodes to let $a[i]$ **float up** to make sure the subtree of $a[i].\text{parent}$ obey the min-heap property. For example, in the min-heap. The time complexity is the same as the height of the complete tree, which is $O(\log n)$.

```

1     def __float__(self, index): # enforce min-heap, leaf-to-root
2         while index // 2: # while parent exist
3             p_index = index // 2
4             print('p', p_index, index)
5             if self.heap[index] < self.heap[p_index]: # a
6                 violation
7                     # swap
8                     self.heap[index], self.heap[p_index] = self.heap[
9                         p_index], self.heap[index]
10                else:
11                    break
12            index = p_index # move up the node
13    def insert(self, val):
14        self.heap.append(val)
15        self.size += 1
16        self.__float__(index = self.size)

```

Pop with Sinking When we pop out the item at root node, or delete any item $a[i]$, an empty spot appears at that position. To maintain the complete binary tree, we first simply use the last item to fill in this spot. However, in a min-heap, the last item will mostly not be the smallest item among the subtree rooted at $a[i]$. The smallest item will appear anywhere in the subtree. We simply do a search starts from node $a[i]$ and compare its value with left and right child. The left and right subtree obey the

min-heap property already, therefore the smallest item is among $a[i]$, left, right. If the node is larger than its smaller child node, we swap the parent with the smaller child, and move our pointer to the smaller child node and repeat the above process until the current node is the smallest among these three nodes. This process is called like sinking down $a[i]$ along the **path $a[i] \rightarrow \text{leaf}$** . Same as the insert in the case of complexity, $O(\log n)$.

```

1  def __sink(self, index): # enforce min-heap, root-to-leaf
2      while 2 * index <= self.size:
3          li = 2 * index
4          ri = li + 1
5          mi = index
6          if self.heap[li] < self.heap[mi]:
7              mi = li
8          if ri <= self.size and self.heap[ri] < self.heap[mi]:
9              mi = ri
10         if mi != index:
11             # swap index with mi
12             self.heap[index], self.heap[mi] = self.heap[mi], self.heap[index]
13         else:
14             break
15         index = mi
16     def pop(self):
17         val = self.heap[1]
18         self.heap[1] = self.heap.pop()
19         self.size -= 1
20         self.__sink(index = 1)
21         return val

```

Now, let us run an example:

```

1 h = Heap()
2 lst = [21, 1, 45, 78, 3, 5]
3 for v in lst:
4     h.insert(v)
5 print('heapify with insertion: ', h)
6 h.pop()
7 print('after pop(): ', h)

```

The output is listed as:

```

1 heapify with insertion: 1 3 5 78 21 45
2 after pop(): 3 21 5 78 45

```

Heapify with Bottom-up Sinking Heapify is a procedure that convert a list to a heap data structure. We have learned the insert procedure. To heapify a list, we can do it through a series of insert iterating through the items in the list and we get an upper-bound complexity of $O(n \log n)$. However, a more efficient way to do it 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 subtree. For leaf nodes, they have no children which satisfies the heap property naturally. Therefore we can jump to the last parent node, which will be at position $a[n//2]$. We apply the sinking process as used in **pop** so that this subtree rooted at current node obeys the heap property. And we iterate through all the parents nodes that is $a[1\dots n//2]$ in reversed order, we can guarantee that final complete binary tree still obeys the heap property. This follows a divide-and-conquer (DP) fashion. Instead of heapify $A[1\dots n]$, we first, heapify $A[n], A[n-1\dots n], A[n-2\dots n], \dots, A[1\dots n]$. The process is shown in Fig. 10.2. With this process, it can give us a tighter upper bound and close to $O(n)$.

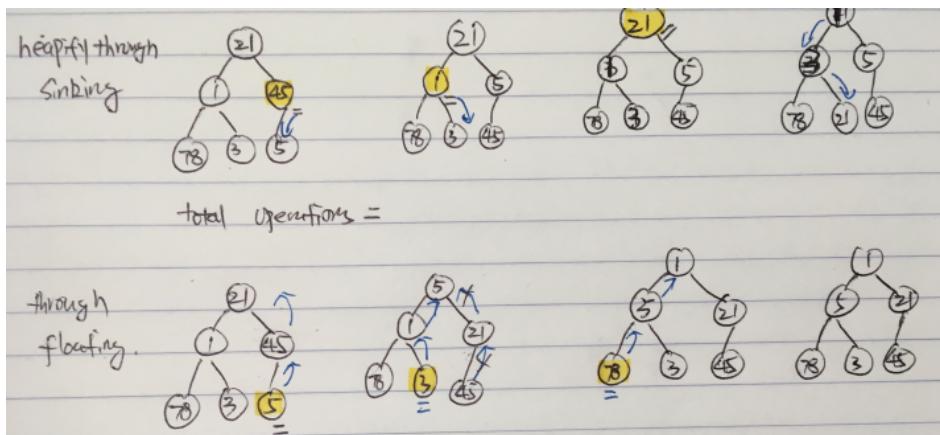


Figure 10.2: Heapify for a given list.

```

1 def heapify_sink(self, lst):
2     self.heap = [None] + lst
3     self.size = len(lst)
4     for i in range(self.size//2, 0, -1):
5         self._sink(i)

```

Now, run the following code:

```

1 h = Heap()
2 h.heapify(lst)
3 print('heapify with heapify:', h)

```

Out put is:

```
1 heapify with heapify: 1 5 21 78 3 45
```



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.

When we are solving a problem, unless specifically required for implementation, we can always use an existent Python module/package. Here, we introduce one Python module: `heapq` that implements heap data structure for us.

10.1.2 Python Built-in Library: `heapq`

heapq: `heapq` is a built-in library in Python that implements relevant functions to carry out various operations on heap data structure. These functions are listed and described in Table 10.2. *To note that `heapq` is not a data type like `queue.Queue()` or `collections.deque()`, it is a library (or class) that can do operations like it is on a heap.* `heapq` has some other

Table 10.1: Methods of `heapq`

Method	Description
<code>heappush(h, x)</code>	Push the value item 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 item 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>heareplace(h, x)</code>	Pop and return the smallest item from the heap, and also push the new item. The heap size doesn't change. If the heap is empty, <code>IndexError</code> is raised. This is more efficient than <code>heappop()</code> followed by <code>heappush()</code> , and can be more appropriate when using a fixed-size heap.
<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.

functions like `merge()`, `nlargest()`, `nsmallest()` that we can use. Check out <https://docs.python.org/3.0/library/heappq.html> for more details.

Min-Heap Now, let us try to heapify the same exemplary list as used in the last section, `[21, 1, 45, 78, 3, 5]`, we use need to call the function `heapify()`. The time complexity of `heapify` is $O(n)$

```

1 '''implementing with heapq'''
2 from heapq import heappush, heappop, heapify
3 h = [21, 1, 45, 78, 3, 5]
4 heapify(h) # inplace
5 print('heapify with heapq: ', h)

```

The print out is:

```
1 heapify with heapq: [1, 3, 5, 78, 21, 45]
```

Here we demonstrate how to use function `nlargest()` and `nsmallest()` if getting the first n largest or smallest is what we need, we do not need to `heapify()` the list as we needed in the heap and pop out the smallest. The step of heapify is built in these two functions.

```
1 ''' use heapq to get nlargest and nsmallest '''
2 li1 = [21, 1, 45, 78, 3, 5]
3 # using nlargest to print 3 largest numbers
4 print("The 3 largest numbers in list are : ", end="")
5 print(heapq.nlargest(3, li1))
6
7 # using nsmallest to print 3 smallest numbers
8 print("The 3 smallest numbers in list are : ", end="")
9 print(heapq.nsmallest(3, li1))
```

The print out is:

```
1 The 3 largest numbers in list are : [78, 45, 21]
2 The 3 smallest numbers in list are : [1, 3, 5]
```

Max-Heap As we can see the default heap implemented in the `heapq` library is forcing the heap property of the min-heap. What if we want a max-heap instead? In `heapq` library, it does offer us function, but it is intentionally hided from users. It can be accessed like: `heapq.__[function]__max()`. Now, let us implement a max-heap instead.

```
1 # implement a max-heap
2 h = [21, 1, 45, 78, 3, 5]
3 heapq._heapify_max(h) # inplace
4 print('heapify max-heap with heapq: ', h)
```

The print out is:

```
1 heapify max-heap with heapq: [78, 21, 45, 1, 3, 5]
```

Also, in practise, a simple hack for the max-heap is to save data as negative. Also, in the priority queue.

More Private Functions

With Tuple/List or Customized Object as Elements Any object that supports comparison (`_cmp_()`) can be used in heap with `heapq`. When we want our item includes information as (priority, task), we can either put it in tuple or list. In the heap, we can change the value of any item just as in the list. However, the problem occurs after the change that the list will violate the heap priority. What we can do is use function such as `_siftdown(heap, 0, len(heap)-1)` (used to implement `heappush`, and called with decreased priority) and `_siftup(heap, 0)` (used to implement `heappop`, and called with increased priority).

Table 10.2: Private Methods of **heapq**

Method	Description
heappush(h, x)	Push the value item onto the heap, maintaining the heap invariant.
heappop(h)	Pop and return the <i>smallest</i> item from the heap, maintaining the heap invariant. If the heap is empty, IndexError is raised.
heappushpop(h, x)	Push item on the heap, then pop and return the smallest item from the heap. The combined action runs more efficiently than heappush() followed by a separate call to heappop().
heapify(x)	Transform list x into a heap, in-place, in linear time.
heareplace(h, x)	Pop and return the smallest item from the heap, and also push the new item. The heap size doesn't change. If the heap is empty, IndexError is raised. This is more efficient than heappop() followed by heappush(), and can be more appropriate when using a fixed-size heap.
nlargest(k, iterable, key = fun)	This function is used to return the k largest elements from the iterable specified and satisfying the key if mentioned.
nsmallest(k, iterable, key = fun)	This function is used to return the k smallest elements from the iterable specified and satisfying the key if mentioned.

```

1 import heapq
2
3 heap = [[3, 'a'], [10, 'b'], [5, 'c']]
4 heapq.heapify(heap)
5 print(heap)
6
7 heap[0] = [6, 'a']
8 print(heap)
9 heapq._siftdown(heap, 0) #similar to remove heap[0], put this item
    at the end
10 print(heap)

```

10.2 Priority Queue

A priority queue is an abstract data type(ADT) and an extension of queue with properties: (1) additionally each item has a priority associated with it. (2) In a priority queue, an item with high priority is served (dequeued) before an item with low priority. (3) If two items have the same priority, they are served according to their order in the queue.

Heap is generally preferred for priority queue implementation because of its better performance compared with arrays or linked list. Also, in Python queue module, we have `PriorityQueue()` class that provided us the implementation. Beside, we can implement priority queue with `heapq` library too. These contents will be covered in the next two subsection.

Applications of Priority Queue:

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.

Implement with heapq Library

The core function is the ones used to implement the heap: `heapify()`, `push()`, and `pop()`. The official document:<https://docs.python.org/2/library/heappq.html> gave the exact implementation. However, we are still going to summarize and organize this information in our book. In order to implement priority queue, our binary heap needs to have the following features:

1. Sort stability: when we get two tasks with equal priorities, we return them in the order as of 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). The entry `count` serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And also, since no two entry counts are the same the tuple comparison will never attempt to directly compare two tasks.
 2. Find a task in the heap, and either remove it or update its priority. Situations like the priority of a task changes or if a pending task needs to be removed. We understand how inconvenient it can be to find the non-root item and update its value. Normally, finding the item is a linear search which takes $O(n)$ and update its value using either `_siftdown()` or `_siftup()` can be $O(\log n)$. The solution is: (1) do not remove the task other than the `pop` operation, but mark it as `REMOVED` instead; (2) to define a dictionary that uses `task` as key and the 3-element list as value. We name it `entry_finder`. When the entry is a list, in the heap that encompass these items will only get pointers. Therefore, we can execute the find/mark as removed operation using `task` as key and do it in the `entry_finder` instead.

Python code:

```
1 from heapq import heappush, heappop, heapify
2 from typing import List
3 import itertools
4 class PriorityQueue:
5     def __init__(self, items: List[List] = []):
6         self.pq = []                                     # list of entries
7         arranged in a heap
8         self.entry_finder = {}                         # mapping of tasks to
9         entries
```

```

8     self.REMOVED = '<removed-task>'      # placeholder for a
9     removed task
10    self.counter = itertools.count()       # unique sequence
11    count
12    # add count to items
13    for p, t in items:
14        item = [p, next(self.counter), t]
15        self.entry_finder[t] = item
16        self.pq.append(item)
17    heapify(self.pq)
18
19    def add_task(self, task, priority=0):
20        'Add a new task or update the priority of an existing task
21        '
22        if task in self.entry_finder:
23            self.remove_task(task)
24        count = next(self.counter)
25        entry = [priority, count, task]
26        self.entry_finder[task] = entry
27        heappush(self.pq, entry)
28
29    def remove_task(self, task):
30        'Mark an existing task as REMOVED. Raise KeyError if not
31        found.'
32        entry = self.entry_finder.pop(task)
33        entry[-1] = self.REMOVED
34
35    def pop_task(self):
36        'Remove and return the lowest priority task. Raise
37        KeyError if empty.'
38        while self.pq:
39            priority, count, task = heappop(self.pq)
40            if task is not self.REMOVED:
41                del self.entry_finder[task]
42                return task
43            raise KeyError('pop from an empty priority queue')

```

Let's run an example with our customized `PriorityQueue` class:

```

1 pq = PriorityQueue(items=[[6, 'task 6'], [5, 'task5'], [19, 'task19']])
2 print(pq.pq)
3 pq.add_task('task 10', 10)
4 print(pq.pq)
5 pq.remove_task('task5')
6 print(pq.pq)
7 pq.pop_task()

```

With output as:

```

[[5, 1, 'task5'], [6, 0, 'task 6'], [19, 2, 'task19']]
[[5, 1, 'task5'], [6, 0, 'task 6'], [19, 2, 'task19'], [10, 3, 'task 10']]
[[5, 1, '<removed-task>'], [6, 0, 'task 6'], [19, 2, 'task19'],
 [10, 3, 'task 10']]
'task 6'

```

Implement with PriorityQueue class

Class `PriorityQueue()` is the same as `Queue()`, `LifoQueue()`, they have same member functions as shown in Table 7.9. Therefore, we skip the semantic introduction. `PriorityQueue()` normally thinks that the smaller the value is the higher the priority is. We use a similar example as above to demonstrate its function.

```

1 import queue
2 pq = queue.PriorityQueue()
3 items=[[6, 'task 6'], [5, 'task5'], [19, 'task19']]
4 for item in items:
5     pq.put(item)
6
7 print(pq.queue)
8 next_job = pq.get()
9 print('processing job:', next_job)
10 print(pq.queue)
```

The output is:

```

1 [[5, 'task5'], [6, 'task 6'], [19, 'task19']]
2 processing job: [5, 'task5']
3 [[6, 'task 6'], [19, 'task19']]
```

If we want to give the number with larger value as higher priority, a simple hack is to pass by negative value. Another more professional way is to pass by a customized object and rewrite the comparison operator: `<` and `==` in the class with `__lt__()` and `__eq__()`. In the following code, we show how to use higher value as higher priority.

```

1 class Job(object):
2     def __init__(self, priority, description):
3         self.priority = priority
4         self.description = description
5         print('New job:', description)
6         return
7     # def __cmp__(self, other):
8     #     return cmp(self.priority, other.priority)
9     '''customize the comparison operators'''
10    def __lt__(self, other): # <
11        try:
12            return self.priority > other.priority
13        except AttributeError:
14            return NotImplemented
15    def __eq__(self, other): # ==
16        try:
17            return self.priority == other.priority
18        except AttributeError:
19            return NotImplemented
20
21 q = Queue.PriorityQueue()
22
23 q.put( Job(3, 'Mid-level job') )
```

```

24 q.put( Job(10, 'Low-level job') )
25 q.put( Job(1, 'Important job') )
26
27 while not q.empty():
28     next_job = q.get()
29     print('Processing job:', next_job.priority)

```

The print out is:

```

1 Processing job: 10
2 Processing job: 3
3 Processing job: 1

```

If we want the priority queue to be able to update the priority of a task, we can apply similar wrapper in the `heapq` section.



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.

Let us take these knowledge into practice with a LeetCode Problem: 347. Top K Frequent Elements (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: to solve this problem, we need to first using a hashmap to get information as: item and its frequency. Then, we need to obtain the top frequent elements. The second step can be down with sorting, or using heap we learned.

Solution 1: Use Counter(). `Counter()` has a function `most_common(k)` that will return the top k most frequent items. However, its complexity will be $O(n \log n)$.

```

1 from collections import Counter
2 def topKFrequent(self, nums, k):
3     return [x for x, _ in Counter(nums).most_common(k)]

```

Solution 2: Use dict and heapq.nlargest(). The complexity should be better than $O(n \log n)$.

```

1 from collections import Counter
2 import heapq
3 def topKFrequent(self, nums, k):
4     count = collections.Counter(nums)
5     return heapq.nlargest(k, count.keys(), key=count.get)

```

We can also use PriorityQueue().

```

1 from queue import PriorityQueue
2 class Solution:
3     def topKFrequent(self, nums, k):
4         h = PriorityQueue()
5
6         # build a hashmap (element, frequency)
7         temp = {}
8         for n in nums:
9             if n not in temp:
10                 temp[n] = 1
11             else:
12                 temp[n] += 1
13         # put them as (-frequency, element) in the queue or heap
14         for key, item in temp.items():
15             h.put((-item, key))
16
17         # get the top k frequent ones
18         ans = [None]*k
19         for i in range(k):
20             _, ans[i] = h.get()
21
22         return ans

```

10.3 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.

10.4 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

Searching Algorithms

Finding a solution to a problem in Computer Science and Artificial Intelligence is often thought as a process of search through the space of possible solutions (state space), either carried on some data structures, or calculated in the search space of a problem domain.

Searching Strategies In this part, we will first learn the basic searching algorithms carried out on explicit data structures in Chapter 11. This will include:

1. General Searching Strategies
2. Linear Search
3. Tree Search
4. Graph Search

In this chapter, we only explain different strategies on explicitly defined data structures to keep it simple and clean. The main purpose is to learn the fundamental concepts and properties to lay the ground for the more advanced algorithms.

Uninformed search vs informed search Searching can be categorized as **uninformed search** (also called **blind search**) and **informed search** (also called **heuristic search**) strategies. The uninformed search means that the strategies have no additional information about states beyond that provided in the problem definition. All they can do is to generate successors and distinguish a goal state from a non-goal state. And we categorize their strategies by the *order* in which nodes are expanded. On the other hand, strategies that know whether one non-goal state is “more promising” than another are the informed search. In this book, we only cover common uninformed search strategies.

Advanced uninformed searching learning the basics and knowing the properties, we can move on to more advanced uninformed searching techniques in Chapter ?? that has better efficiency. The content will include:

1. Advanced Linear Searching such as Binary Search and Two-pointer technique;
2. Recursive Backtracking;
3. Bidirectional BFS.

Complete Search and **partial search** are the two main branches in the Searching paradigm.

Complete search is one that guarantees that if a path/solution to the goal/requirement exists, the algorithm will reach the goal given enough time, this is denoted as *completeness*. Complete search is thought of as a *universal solution* to problem solving. On the other hand, Partial Search a.k.a Local Search will not always find the correct or optimal solution if one exists because it usually sacrifice completeness for greater efficiency.

In this part of this book, we will learn the Complete Search instead of the partial search due to its practicity solving the LeetCode Problems. The name “Complete Search” does not necessarily mean they are not efficient and a brute force solution all the time. For instance, the **backtracking** and **Bi-directional Search** they are more efficient than a brute force exhaustive search solutions.

Complete Search algorithms can be categorized into:

- Explicit VS Virtual Search Space: Explicit complete search is carried on data structures, linear structures or non-linear data structures like graph/ trees. In Explicit Search, the search space size is the size of the targeting data structure size. We will need to find a sub-structure of the given data structure. Virtual space based search is to find a set of value assignments to certain variables that satisfy specific mathematical equations/inequations, or sometimes to maximize or minimiaze a certain function of these variables. This type of problems is known as **constraint satisfaction problem**. Such as backtracking an optimized search algorithms for virtual space.
- Linear VS Non-linear Complete Search: Linear search checks every record in a linear fashion, such as sliding window algorithm, binary search, sweep linear. On the other hand, Non-linear Search is applied on non-linear data structures and follows graph fashion.
- Iterative VS Recursive Search: For example, most linear search is iterative. Breath-first-search for graph and level-by-level search for trees are iterative too. Recursive Search are algorithms implemented with recursion, such as Depth-first-search for graph, or DFS based tree traversal, or backtracking.

How to Learn Complete Search? Up till now, we have already learned the explicit complete search carried out on different data structures in Part III. In this part, we will complete the complete search topic with more advanced and efficient searching algorithms applied either on real data structures or result space. Also, this part will give us more examples of how to apply the algorithm design methodology as divide and conquer, use the basic data structures we learned before to design **complete search** algorithms with efficiency and elegance.

Organization of Complete Search This part follows the same organization as of Part III, it is composed of linear search or non-linear search. For each type of algorithm, we will explain how to use it on cases like: explicit or virtual search space.

- Linear Search (Chapter 12) which describes the common algorithms that carries on Linear data structures: Array, Linked List and String.
- Non-linear Search (Chapter 13) encompasses the most common and basic search techniques for graph and tree data structures. The two most basic search techniques: Breadth-first-search and Depth-first-search serves as the basis to the following more advanced graph algorithms in the next chapter.
- Advanced Non-linear Search (Chapter ??) includes more advanced concepts of graph and more advanced graph search algorithms that solve common problems defined in graph. The problems specifically we include are: Connected Components, topological sort, cycle detection, minimum spanning trees and shortest path related problems.

In this part, for graph algorithms, we will only cover medium level, and leave out the more advanced ones in Part VII.

Searching Methodology Before we head to more specific searching algorithms, we shall define the searching algorithm design methodology more clearer:

11

Basic Searching

In this Chapter, we focus on learning general search strategies include: breath-first, depth-first, and priority-first searching. The basic concepts will be explained in Section. 11.1.

After this, we head out to apply these strategies given explicit data structures for linear data structure in Sec. 11.2, Tree data structure in Sec. 11.3 which are also called tree traversal algorithms, Graph data structures in Sec. 11.4. Along the applications, we will analyze them in terms of time and space complexity, completeness and optimality.

At the end, we will compare the difference of applying the searching strategies on the tree and graph data structures mainly how they would affect the completeness and optimalties.

11.1 General Searching Strategies

As we have mentioned in Chapter ??, there are generally three different searching strategies according to the orders in which nodes are expanded: breath-first search, depth-first search and priority-first search. In this section, we focus on the concepts instead of the exact implementation which varies to different data structures and will be detailed in the remaining sections.

Also, it is important to make clear of the concept of the tree-search and graph search. Tree search can happen on either a tree or a graph data structure.

Tree Search on Tree Data Structure On the tree data structure, because there will be one and only one path between any two nodes in the

tree, so normally we do not need to check the repeated nodes.

Graph or Tree Search on Graph Data Structure On the graph data structure, first, there might be more than one path between some two nodes in the graph. Second, there might have loop exist. Therefore, the common strategy in graph search is to use a set data structure to check repeated nodes, that is graph search will visit each node one and only once. The visiting order of the nodes can be connected as a search tree(that connects all n vertexs with $n - 1$ edges). This avoids the redundant paths and the loop.

We can also treat the graph as a tree, that the source node is a root, and any neighboring nodes will be children. In the graph, whenever to judge if a neighboring node is a child or not, we check if we have already visited this node from our path (it can not be our parent or grandparent node). So, this is a tree-search version on the graph data structure.

11.1.1 Breath-first Search

Breath-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. The process is shown in Fig. 11.1.

Implementation Breath-first search can be implemented iteratively using FIFO queue data structure as the storage of the frontiers. With the first-in, first out rule, new nodes (which are always deeper than their parents) go to the back of the queue, and the old ones, which are shallower than the new ones, get expanded first.

Properties

For breath-first search,

Completeness Breath-first search is *complete*—if the shallowest goal node is at some finite depth d , breath-first search will eventually find it after generating all shower nodes (provided that branching factor b is finite).

Optimality Breath-first can always find the *shallowest* path to a goal node. Because that as soon as a goal node is generated, we know it is the shallowest goal node because all shower nodes must have been generated already and failed the goal test (not the goal node in an explicit graph). (prove with something in introduction to algorithms).

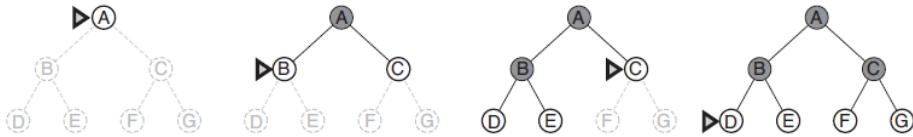


Figure 11.1: Breath-first search on a simple search tree. At each stage, the node to be expanded next is indicated by a marker.

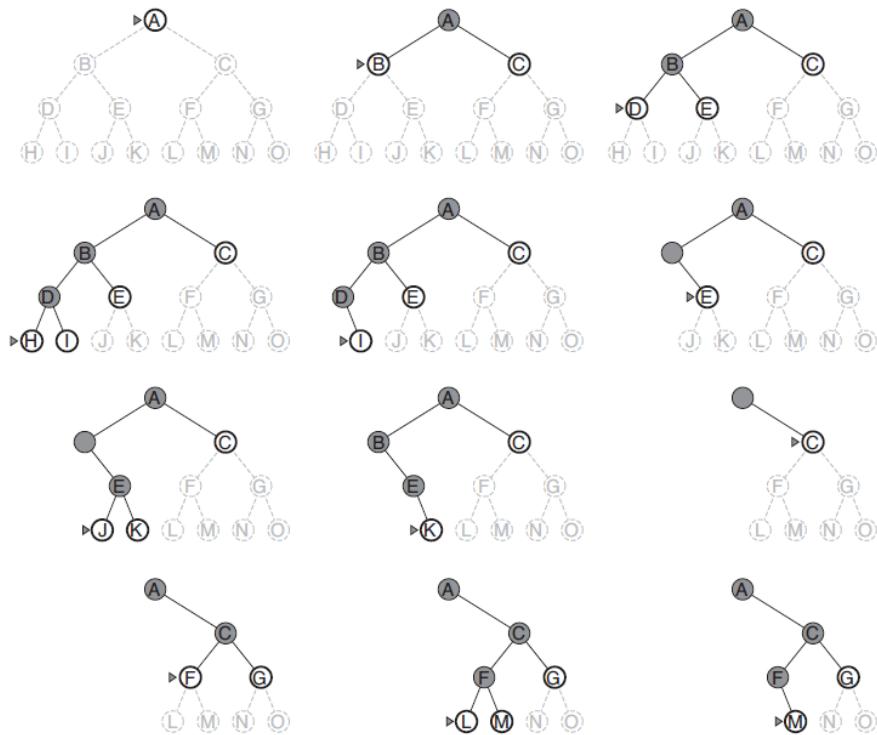


Figure 11.2: Depth-first search on a simple search tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory as node L disappears. Dark gray marks nodes that is being explored but not finished.

When the path cost is a nondecreasing function of the depth of the node, then breath-first search is optimal. The most common such scenario is that all actions have the same cost as 1. However, if the condition can not be satisfied, even if breath-first search can guarantee that we find the shallowest path, but this is not necessarily the *optimal*.

So far, both the completeness and optimality looks promising of breath-first search, we can analyze its complexity in terms of time and space.

Time Complexity the time complexity is bounded by the size of the state space, which is essentially a graph, which makes our complexity bound to be

$O(|V| + |E|)$. A tighter bound of the time complexity of breath-first search is the same as its nodes in the search tree. Starting from root node, assume an uniform tree which has equal branch factor for all nodes b . The nodes at each level i will be b^i , $i \in [0, d]$. Now, suppose that the solution is at depth d . In the worst case, it is the last node generated at d -th level. Then the total number of nodes is :

$$\sum_{i=0}^d b^i = O(b^d) \quad (11.1)$$

Space Complexity For breath-first search, in some condition such in the graph, we need to keep two sets: *explored set* and *frontier list*. The space complexity gets maximum at the last level, where there will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier. If it is a tree, we do not need to keep the explored set to avoid loops, therefore, the space complexity comes from the space taken by the frontier list. An exponential complexity bound such as $O(b^d)$ is actually scary.

(table time and memory requirements for breath-first search)

Two lessons can be learned from this given table. First, the memory requirements are a bigger problem for breath-first search than is the execution time. The second lesson is that time is still a major factor. If your problem has a solution at depth 16, then it will takes about 350 years for breath-first search to find it. Knowing the time complexity of search strategies can guide us to find more efficient algorithms to solve the real problems.

11.1.2 Depth-first Search

Depth-first search on the other hand, will always expand the *deepest* node in the current frontier of the search tree. The progress is illustrated in Fig. 11.2. First, our frontier is only A, so we expand A. Then B and C become frontier. They would have the same depth, so expand any one of them would be reasonable. Here, we expand the first one if they share the same depth. We expand B, and add D and E into the frontier, then expand D, and add H, I. Expand H, finished and we remove it from memory. Next expand I. Similarly and remove I from memory. As we see, depth-first search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search “back up” to the next deepest node in the frontier and repeat the process again.

Implementation Depth-first can be both implemented recursively and iteratively with LIFO queue which is also called stack.

If using a stack, we can have the most recently node in the frontier chosen for expansion. This must be the deepest unexpanded node because

it is one deeper than its parent, and at that moment expanding the parent, the parent was the deepest.

It is also very common to implement depth-first search with a recursive function that calls itself on each of its children in turn. This implementation can be think of using divide-and-conquer. First we have $\text{DFS}(A)$, to traversal all of A, we visit A, then we send out two agents to traverse B and C. We can think of it as:

$$\text{DFS}(A) = \text{visit}(A) + \text{DFS}(B) + \text{DFS}(C) \quad (11.2)$$

The recursive function will return if we have a None node or a leaf node which has no further successors to expand.

Properties

The properties of depth-first depend strongly on whether the graph-search or tree-search version is used. We will explain the basic concepts here and will later compare more with the exact Python implementation.

Completeness and Optimality In the graph-search version, if we do not avoid repeated nodes, we will get stuck into loops, which will make the depth-first search *incomplete*. If we avoid repeated nodes, we will avoid this situation, then our graph-search version will be *complete* in finite state spaces because it will eventually expand every node. However, the limitation of only expanding each node once will make the graph-search version *nonoptimal*. For example, to find the path to node C, depth-first search might end up exploring the entire left subtree before it find the goal on the right side. Most importantly, if our task is to find the shortest path to reach to a node, depth-first search in the graph will have problem with it, because it only connects the vertex in the graph as a tree, and so to each vertex, depth-first graph search will only get one path from the source to the target. This one path would not be the shortest one on most chances.

In the tree-search version, if the original data structure is a graph, then it is necessary for us to check new states against those on the path from the root to the current node to avoid infinite loop. Therefore, with the path checking, our tree-search dfs is *complete*. However, it is *nonoptimal*. Say the task of finding the shortest path, tree-search dfs cannot just stop when the goal node is found, it needs to keep searching till it ended then by comparing possible multiples redundant paths to the same goal, it can find itself the shortest.

Time Complexity Same as breath-first search, the time complexity is bounded by the size of the state space, which is essentially a graph, which makes our complexity bound to be $O(|V| + |E|)$. In the tree-version, it may

generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node; this can be even much larger than the size of the state space. Compared with the time complexity of bfs, m can be much larger than d .

Space Complexity So far, we might think, wait a minute, depth-first search might not be complete, definitely nonoptimal, and even has worse time complexity compared with breath-first search, why would we still need it? The reason is the space complexity. For the graph search version, it has no space advantage, but a depth-first search tree needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. For a state space with branching factor b and maximum depth m , depth-first search requires storage of only $O(bm)$ nodes.

Because of this single reason, we would see in the remaining searching content of the book that how often a DFS can be used for different tasks, such as Cycle Check, Topological sort (might because the special order of dfs), backtracking. (need testify) (This has led to the adoption of depth-first tree search as the basic workhorse of many areas of AI, including constraint satisfaction (Chapter 6), propositional satisfiability (Chapter 7), and logic programming (Chapter 9). For the remainder of this section, we focus primarily on the tree search version of depth-first search.)

1.

11.1.3 Priority-first Search

In breath-first search, it is optimal because it always expand the shallowest unexpanded node, thus finding the shortest path from a given source to a target.

11.2 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 sequentiall and every item is checked until a target is found. It just need a for/while loop, which gives as $O(n)$ as time complexity, and no extra space needed.

Implementation The implementation is straightforward:

```

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 because of its efficiency compared with other searching methods we have learned (hashmap) or will learn (binary search, two-pointer search).

11.3 Tree Traversal

11.3.1 Depth-First Tree Traversal

Implementation of Free Tree From the view of the free tree, the tree traversal can be implemented with DFS with one node be chosen as a source. As shown in Fig. 15.1, after we delete two edges from the graph, we have a tree structure. The original DFS traversal path in the graph is 1, 2, 4, 6, 5, 3, which actually compose a tree, we can this a DFS traversal tree. Now, let us implement a general purpose DFS traversal in tree, it would be the same as of the DFS and with no need of the state record.

```

1 def dfs(t, s):
2     '''implement the dfs recursive of tree'''
3     print(s)
4     for neighbor in t[s]:
5         dfs(t, neighbor)
6     return

```

Let us run the following example:

```

1 def dfs(t, s):
2     '''implement the dfs recursive of tree'''
3     print(s)
4     for neighbor in t[s]:
5         dfs(t, neighbor)
6     return

```

It actually has the same output as the DFS traversal in the graph.

Tree traversal for the rooted tree However, for a rooted tree, which has a singled out starting point, we have three typical mutants of dfs traversal: they are pre-order, in-order, post-order based on the order of visiting the root node with the following example shown in Fig. 11.3.

- (a) Inorder (Left, Root, Right) : 4, 2, 5, 1, 3
- (b) Preorder (Root, Left, Right) : 1, 2, 4, 5, 3
- (c) Postorder (Left, Right, Root) : 4, 5, 2, 3, 1

For Breadth First or Level Order Traversal : 1, 2, 3, 4, 5.

First, for the unrooted tree, the implementation is a simplified version of graph DFS traversal with given a starting node. And the traverse process we do not need to record its visiting states.

More likely, we need to In this section, we will show how to implement DFS based three types of tree traversal: PreOrder, InOrder, and PostOrder.

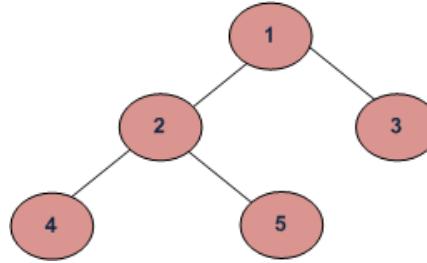


Figure 11.3: Binary Tree

Recursive with Divide and Conquer

Here, we first use PreOrder traversal as an example. To make it easier to understand, the queen has a target, when she is given a job, she assigns it to two workers – A and B to collect the result from left subtree and the right subtree, which we get $A = [2, 4, 5], B = [3]$. Then the final result = the result of the $queue + left + right = [1, 2, 4, 5, 3]$.

```

1 def PreOrder(root):
2     if root is None:
3         return []
4     res = []
5     # divide: into handling left subtree and right subtree
6     left = PreOrder(root.left)
7     right = PreOrder(root.right)
8     # conquer: current node
9     res = [root.val] + left + right
10    return res
11 print(PreOrder(root))
12 # output
13 # [1, 2, 4, 5, 3]
  
```

Similarly, the recursive code for the InOrder Traversal and PostTraversals:

```

1 def InOrder(root):
2     if root is None:
3         return []
4     res = []
5     left = InOrder(root.left)
6     #print(root.val, end=',')
7     right = InOrder(root.right)
8     res = left + [root.val]+ right
9     return res
10
11 def PostOrder(root):
12     if root is None:
13         return []
14     res = []
15     left = PostOrder(root.left)
16     #print(root.val, end=',')
17     right = PostOrder(root.right)
18     res = left + right + [root.val]
  
```

```

19     return res
20 print(InOrder(root))
21 print(PostOrder(root))
22 # output
23 #[4, 2, 5, 1, 3]
24 #[4, 5, 2, 3, 1]

```

Iterative with DFS and stack

Since the PreOrder, InOrder, and PostOrder tree traversal are all depth-first-search, we can use stack to save the result. This iterative implementation is better than the recursive version, because the memory we use here is the heap memory = memory size. While for the recursive version, it uses the stack memory = processing memory, so it is easier to run out of memory.

PreOrder Iterative . To make preorder traversal is a straightforward DFS, which visits the root at first, and then the children. We can use refer to DFS in the last set iterative, which visits the root at first, which is the element popped out of the stack every time. Note: because the stack is FILO, if we want to visit left subtree at first, we need to push the right subtree at first. Thus, preorder traversal's iterative implementation is a very standard DFS with stack.

```

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     return res
14 # output
15 #[1, 2, 4, 5, 3]

```

PostOrder Iterative . Need to explain better!!!

```

1 def postorderTraversal(self, 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]

```

InOrder Iterative . In the inorder, we need to print out all the left subtree first, and then the root, followed by the right. The process is as follows:

- 1) Create an empty stack S.
- 2) Initialize current node as root
- 3) Push the current node to S and set current = current->left until current is NULL
- 4) If current is NULL and stack is not empty then
 - a) Pop the top item from stack.
 - b) Print the popped item, set current = popped_item->right
 - c) Go to step 3.
- 5) If current is NULL and stack is empty then we are done.

```

1 def InOrderIterative(root):
2     if root is None:
3         return []
4     res = []
5     stack = []
6     current = root
7     while current:
8         stack.append(current)
9         current = current.left
10
11    while stack:
12        tmp = stack.pop()
13        res.append(tmp.val)
14        current = tmp.right
15        while current:
16            stack.append(current)
17            current = current.left
18
19    return res

```

Another way to write this:

```

1 def inorder(self, root):
2     cur, stack = root, []
3     while cur or stack:
4         while cur:
5             stack.append(cur)
6             cur = cur.left
7         node = stack.pop()
8         print(node.val)
9         cur = node.right

```

11.3.2 Breath-first Tree Traversal

Level Order Tree Traversal Because level order tree traversal is intuitively a breath-first-search, within which we use queue data structure to implement it.

```

1 def LevelOrder(root):
2     if root is None:
3         return []
4     q = [root]
5     res = []
6     while q:
7         new_q = []
8         for n in q:
9             res.append(n.val)
10            if n.left:
11                new_q.append(n.left)
12            if n.right:
13                new_q.append(n.right)
14        q = new_q
15    return res
16 print(LevelOrder(root))
17 # output
18 # [1, 2, 3, 4, 5]
```

11.4 Searching on Graph

Visiting States Because in graph, it is reasonable to expect it contains cycles. In our example, we have a cycle $[0, 1, 3, 4, 2, 0]$. Therefore, in the graph search, it is a necessity to avoid traversing a cycle will make the program running nonstop. The solution is during the search process, we mark states for each vertices. In the graph search, there are three possible states:

1. WHITE: (False, -1), which is the initial state of each vertex in the G , and has not been visited yet.
2. BLACK: (True, 1), which marks that the node is fully visited and complete. More specifically that all vertices adjacent to them are nonwhite.
3. GRAY: (0), this is an intermediate state between WHITE and BLACK, which is ongoing, visiting but not completed that we have not yet checked out all its incident edges. More specific, gray vertex might have adjacent vertices of all three possible states.

We can define a STATE class for convenience.

```
class STATE:
    white = 0
```

```
gray = 1
black = 2
```

11.4.1 Breadth-first Search (BFS)

Given a graph $G = (V, E)$, and a *source* vertex s , the aim of Breadth-first search is to explore the edges of G to discover all vertices that are reachable from the source s .

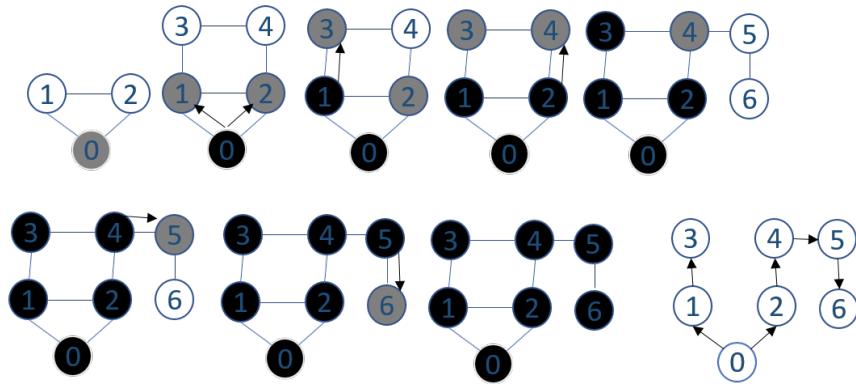


Figure 11.4: The process of Breadth-first-search. The black arrows denotes the relation of u and its not visited neighbors v . All these edges constructs a breath-first-tree. The visiting orders of BFS starting from vertex 0 is $[0, 1, 2, 3, 4, 5, 6]$.

The principle of the breath-first-search as how it is named, that is it visit vertices that are reachable from the source in the order of distance from the source. More specifically, in the example, we are given vertex 0 as the source. And first, we visit its neighbors 1, 2 since they are the closed nodes. The edge $(0, 1), (0, 2)$ are added to the BFS tree. Next, move to 0's first neighbor 1, and visited 1's unvisited neighbors, 3. Next, visit 2's unvisited neighbors 4, and so on. The whole process is depicted in Fig. 11.4.

Basic Implementation

Level by Level Implementation Other than using a `queue` and visit each node one by one, we can use a `list` and save all nodes that locate at the same level, meaning the same distance from the source. For our example, start with node 0, so the list is initialized as `[0]`. And each iteration we visit the unvisited neighbors of all nodes in the current list and during this process, it uses another temporary list to save the newly added neighbors. After the iteration, it replaces the original list with the temporary list. So, first we visit all neighbors of 0, which is 1 and 2, then our list becomes `[1, 2]`, we then visit neighbors of them, that is 3 and 4, thus the list becomes

[3, 4], and so on. We distinguish these two methods as **node by node** and **level by level** process. The advantage of doing so is easy to track distance because all nodes in the list from source and also they are all sharing the same distance.

```

1 def bfs_level(g, s):
2     '''level by level bfs'''
3     v = len(g)
4     state = [False] * v
5
6     orders = []
7     lst = [s]
8     state[s] = True
9     d = 0 # track distance
10    while lst:
11        print('distance ', d, ': ', lst)
12        tmp_lst = []
13        for u in lst:
14            orders.append(u)
15            for v in g[u]:
16                if not state[v]:
17                    state[v] = True
18                    tmp_lst.append(v)
19        lst = tmp_lst
20        d += 1
21    return orders

```

If we run the same example with `print(bfs_level(al, 0))`, we will have the same path as the first implementation.

```

distance 0 : [0]
distance 1 : [1, 2]
distance 2 : [3, 4]
distance 3 : [5]
distance 4 : [6]
[0, 1, 2, 3, 4, 5, 6]

```

For the implementation of BFS, it is a good chance for us to see the usage of `queue` data structure in practice. Because the property of queue that states “first come first served” aligns with the principle of breath-first-search and this keeps the distance of the visiting’s node in increase order from the source. We will start with simple implementation and see how it works. Then start to discuss about more advanced properties of BFS and implement.

Implementation with Queue In this version, we only track the ordering of visiting nodes. There are two lists used: `orders` to record nodes once it becomes gray, and `complete_orders` to record once it becomes black.

```

1 def bfs(g, s):
2     '''simplified bfs'''
3     v = len(g)

```

```

4     colors = [STATE.white] * v
5
6     q, orders = [s], [s]
7     complete_orders = []
8     colors[s] = STATE.gray # make the state of the visiting node
9     while q:
10        u = q.pop(0)
11
12        for v in g[u]:
13            if colors[v] == STATE.white:
14                colors[v] = STATE.gray
15                q.append(v)
16                orders.append(v)
17            # complete
18            colors[u] = STATE.black
19            complete_orders.append(u)
20    return orders, complete_orders

```

Call the above function with `print(bfs(al, 0))`, we have the following result:

```
([0, 1, 2, 3, 4, 5, 6], [0, 1, 2, 3, 4, 5, 6])
```

Simplify the States We see when we saving the nodes when they first turn gray or turn black, we end up with the same ordering. This reveals that in breath-first-search, it is not necessary for us to distinguish the state gray and black. Thus having two states will be enough for most BFS based algorithms. Using three states just help us understand and distinguish the search procedure. Therefore, in the following code, we only track two states: visited or unvisited.

At first, all vertices are having `False` to represent the unvisited state, then we have two ways to mark the node as visited: either when we enqueue the node or dequeue the node.

Complexity Analysis Because in the process of BFS, each vertex is enqueued and dequeued exactly one time, this process takes $O(|V|)$. Because at each iteration, it scans each edge exactly once too, this takes $O(|E|)$. Sums up we have the total time complexity of $O(|V| + |E|)$.

Shortest Paths and Advanced Implementation

Shortest paths from s to t can be found using BFS. The shortest-path distance is defined as the minimum number of edges among all paths from s to t . The correctness and proof¹ of this is skimmed. But we can reason with level by level BFS: we start with source, which the shortest distance to itself is 0. Then, for all its neighbors, the shortest distance will be 1. Then for

¹Refer to *Introductions to Algorithms*

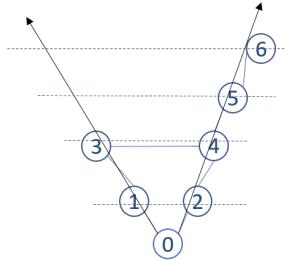


Figure 11.5: Visualize the BFS in level by level fashion.

node 3, 4, to find its shortest distance, it to connect them with neighboring nodes that have the shortest distance. Actually, in the sense of finding the shortest paths, BFS is a greedy approach. Therefore, tracking the distance of each vertex from the source and be able to print out the shortest path between source and a target vertex at the same time is preferred.

Therefore, we can update our above code with two features: distance list and predecessor list. To be able to print out the shortest path between source and any reachable target, we need to track *predecessor* of each node. u is the predecessor of v only if there is edge (u, v) and v is the unvisited neighbor of u .

```

1 def bfs(g, s):
2     '''node by node bfs using queue'''
3     v = len(g)
4     state = [False] * v
5
6     # allocate space for the predecessor list and colors
7     pi = [None] * v
8     state[s] = True # make the state of the visiting node
9     dist = [0] * v
10
11    q, orders = [s], [s]
12    while q:
13        u = q.pop(0)
14        for v in g[u]:
15            if not state[v]:
16                state[v] = True
17                pi[v] = u # set the predecessor
18                dist[v] = dist[u] + 1
19                q.append(v)
20                orders.append(v)
21    return orders, pi, dist

```

Print Shortest Path To be able print the path of s and t we can start with t and traverse back to s through the predecessor. That is we first check out t 's predecessor if it has one, and then the predecessor's predecessor, and so on. There are two ways to do it: recursive and iterative.

In the recursive solution, we first call (s, t) , and then $(s, \pi[t])$ up till the base case where the source and the target is the same. Because we are visiting in reverse order, therefore, we first save the path of the base case, and the predecessor in the path after the recursive call.

```

1 def get_path(s, t, pi, path):
2     '''recursive'''
3     if s == t:
4         path.append(t)
5         return
6     elif pi[t] is None:
7         print('no path from ', s, ' to ', v)
8     else:
9         get_path(s, pi[t], pi, path)
10    path.append(t)
11    return

```

In the iterative solution, we use a `while` loop to visit the predecessor till we meet the source. The path will at last be reversed.

```

1 def get_path(s, t, pi):
2     '''iterative'''
3     p = t
4     path = []
5     while p != s:
6         path.append(p)
7         p = pi[p]
8     path.append(s)
9     return path[::-1]

```

Breath-first Tree The procedure of BFS builds a breath-first tree with the source as root. As shown in Fig. 11.4, all the edges with arrows compose the bfs tree. The tree contains all vertices reachable from s , if we denote nodes as V_t and the edges are from each node's predecessor to this node, denotes as $E_t = (\pi[V_t], V_t), V_t \neq s$. The subgraph of (V_t, E_t) is called the **Predecessor Subgraph**, in BFS, this makes a tree because the number of edges is one less compared with number of vertices. We name this tree breadth-first tree.

Multiple Starts Also, it is necessary for us to know how to access any node starts from multiple nodes.

```

1 #implement using a queue
2 def BFSLevel(starts):
3     q = starts # a list of nodes
4     #root.visited = 1
5     while q:
6         new_q = []
7         for node in q:
8             for neig in node.adjacent:
9                 if not neig.visited:
10                    neig.visited = 1

```

```

11           new_q.append( neig )
12 q = new_q

```

Applications

BFS usually will be applied in situations with matrix, graph, or tree. The common problems that can be solved by BFS is to problems that just need one solution: the best one like getting the minimum path. And usually not be applied to get all possible paths from source to destination. Also, there are more advanced graph algorithms that we will learn later uses the breath-first-search as archetype, such as Prim's minimum-spanning-tree algorithm and Dijkstra's single-source-paths algorithm.

11.4.2 Depth-First-Search (DFS)

Unlike the BFS that traverse nodes reachable to the source in a level by level fashion, Depth-first Search starts from a given source, and follows a single path in the graph to go as “far” as possible to visit unvisited nodes until (1) it meets a vertex that has no edge out; or (2) no unvisited adjacent vertices or say white vertices. Then it “backtracks” to its predecessor and start the above process again. Same as BFS, DFS will discover all vertices that are reachable from the given source too.

Depth-first Tree Search

First, before we head off to discuss DFS in the graph, let us see how it goes if we directly apply the tree-based DFS search on the graph and see what will happen. If repeat node in the tree path is included, then the path will be looped and the program will never end. This will cause the search to be incomplete (will not find get out of the loop and find other path). Therefore, we include code shown in line 3 to avoid this to happen.

```

1 def treeDFS(g, s, path):
2     for v in g[s]:
3         if v not in path:
4             path.append(v)
5             print('before: path', path)
6             treeDFS(g, v, path)
7             path.pop()
8             print('after: path', path)

```

We can call this function, with `path=[0]`, `treeDFS(al, 0, path)`. We will see the traversal of this tree based DFS is shown in Fig.11.6.

Depth-first Graph Search

Use the example shown in Fig. 11.7. We start from 0, mark it gray, and visit its first unvisted neighbor 1, mark 1 as gray, and visit 1's first unvisited

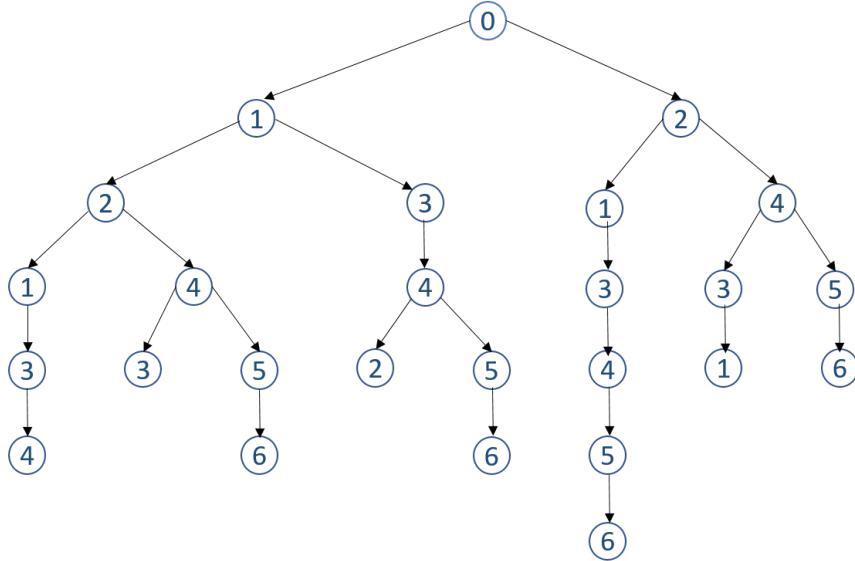


Figure 11.6: The search tree using depth-first tree search

neighbor 2, then 2's unvisited neighbor 4, 4's unvisited neighbor 3. Then we are at the fifth and the sixth subgraph in the first row. Because for 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. In this figure, red arrow marks the backtrack edges. And then we keep the process till 6 become gray. Because 6 has no edge our any more, the state will be complete after 3. Then backtrack to 5, 5 become black, backtrack to 4, then 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, 6], and for the black is [3, 6, 5, 4, 2, 1, 0]. Therefore, it is necessary to distinguish the three states.

The definition of DFS aligns well with recursion programming. Following this, we first discuss the recursive implementation for simplicity. Also, it is not

Basic Recursive Implementation

The three states change the same way as in BFS. It starts with white, and turns into gray when it is visiting and be blackened once all of its neighbors are visited. The only difference for each neighboring node, we goes deeper or say farther by the recursive call to its neighbor. The state turns to back when it is all of its neighbors complete the recursive process :

```

1 def dfs(g, s, colors, orders, complete_orders):
2     colors[s] = STATE.gray
3     orders.append(s)
  
```

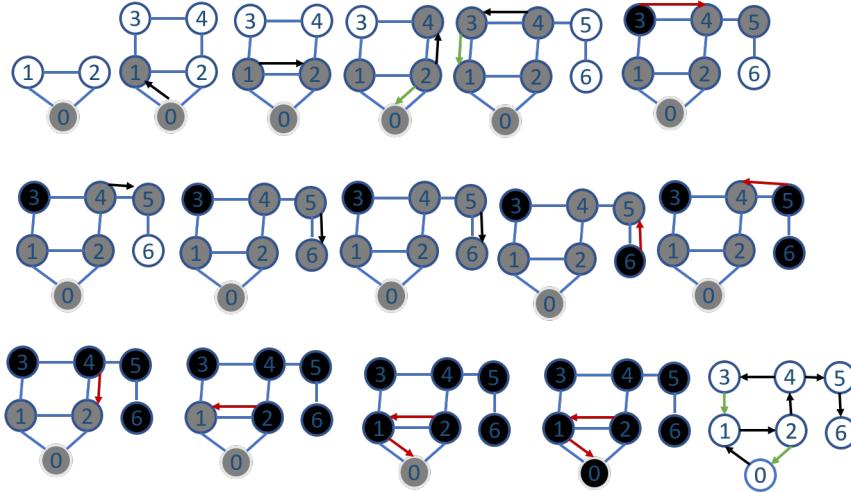


Figure 11.7: The process of Depth-first-search. The black arrows denotes the relation of u and its not visited neighbors v . And the red arrow marks the backtrack edge.

```

4     for v in g[s]:
5         if colors[v] == STATE.white:
6             dfs(g, v, colors, orders, complete_orders)
7     # complete
8     colors[s] = STATE.black
9     complete_orders.append(s)
10    return

```

Now, we try to call the function with the same source of 0:

```

1 # initialization
2 v = len(al)
3 orders, complete_orders = [], []
4 colors = [STATE.white] * v
5 dfs(al, 0, colors, orders, complete_orders)
6
7 print(orders, complete_orders)

```

Now, the `orders` and `complete_orders` will end up differently:

```
[0, 1, 2, 4, 3, 5, 6] [3, 6, 5, 4, 2, 1, 0]
```

Depth-first Tree Search VS Depth-first Graph Search To compare with depth-first tree search, we can print out the path and the search tree of the searching process, which is shown in Fig. 11.7 at the last.

```

1 def dfs(g, s, colors, path):
2     colors[s] = STATE.gray
3     for v in g[s]:
4         if colors[v] == STATE.white:

```

```

5     path.append(v)
6     print('path:', path)
7     dfs(g, v, colors, path)
8     path.pop()
9     # complete
10    colors[s] = STATE.black # this is not necessary in the code,
11        just to help track the state
12    return

```

Now, what is the difference? The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used. The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node. The tree-search version, on the other hand, is not complete if we do not check repeat nodes on the path. Depth-first tree search can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node and to be complete; this avoids infinite loops in finite state spaces but does not avoid the proliferation of redundant paths. For depth-first graph search, however, for each goal vertex, it will only find one path. We can say both version of search is complete which is defined that if there are solutions exist, complete search will find one.

Both version of search is *nonoptimal*. For example, if the task is to find the shortest path from source 0 to target 2. The shortest path should be 0->2, however depth first graph search will return 0->1->2. For the search tree using depth-first tree search, it can find the shorest 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.

Multiple Starts

Complexity Analysis Same to the BFS, we can use aggregate analysis. The DFS cover all edges and vertices, which makes the time complexity of $O(|V| + |E|)$. 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.

Universal Properties of DFS in Undirected and Directed Graph

Depth-first Tree Similar to the Breath first tree, the source will be the root. The same concept of the predecessor subgraph. The edges that appear in the depth-first tree are called tree edges

Tree Edge and Back Edge The DFS prcess helps us to categorize the edges in the graph: in either the directed or undirected graph, we can use it to distinguish between the following two edges.

1. Tree Edges: tree edges are edges in the depth-first tree. The key is that we first explore an edge (u, v) , the color of the vertex v tells us about the edge. If the state of the vertex v is white, it indicates that this is tree edge.
2. Back Edges: back edge is edge (v, u) that connects a vertex v back to its ancestor u . If the state of the vertex u is gray, then it is a back edge.

In the undirected graph, the number of tree and back edges are double of the edge set. And every edge of G is either a tree edge or a back edge. Back edge is not related to backtrack.

Parenthesis Structure In DFS, the discovered time and the finish time has the parenthesis structure. To example this, let us look at the above example, we use a static variable t of function to track the time. dt and ft is used to record the discovered and finished time. Now our dfs is defined as:

```

1 def dfs(g, s, colors, dt, ft):
2     dfs.t += 1 # static variable
3     colors[s] = STATE.gray
4     dt[s] = dfs.t
5     for v in g[s]:
6         if colors[v] == STATE.white:
7             dfs(g, v, colors, dt, ft)
8     # complete
9     dfs.t += 1
10    ft[s] = dfs.t
11    return

```

From the discover time and finish time list, we can generate a new list of merged order merge_orders that arrange the node along the time. And we print out the node the first time it appears as ' $(v,$ ' and second time as ' $v)$ '.

```

1 # initialization
2 v = len(al)
3 dt, ft = [-1] * v, [-1] * v
4 colors = [STATE.white] * v
5 dfs.t = -1
6 dfs(al, 0, colors, dt, ft)
7
8 merge_orders = [-1] * 2 * v
9
10 for i, t in enumerate(dt):
11     merge_orders[t] = i
12
13 for i, t in enumerate(ft):
14     merge_orders[t] = i
15
16 print(merge_orders)

```

```

17 nodes = set()
18 for i in merge_orders:
19     if i not in nodes:
20         print('(', i, end = ', ')
21         nodes.add(i)
22     else:
23         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 )
3

```

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.

Iterative Implementation

In this book, we give two different versions of iterative implementations: (1) DFS, but not preserving the same discovering order (1) keep only the discovering order (2) keep both the discovering order and the finishing order. With these two iterative implementations, we have flexibility to pick one that work out the best.

Method 1: Not preserving the discovering ordering The states that the node has first discovered last completed property, which means using a stack data structure we are able to implement iterative DFS. For example, we first have stack [0], then we put all of its unvisited vertices in [1, 2], then we deal 2, and put all of 2's white neighbors in [1, 2, 4], then 4, [1, 2, 3, 5]. Then 5, [1, 2, 3, 6]. Then it is 6, 3, 2, 1. Therefore, the ordering of each vertex first in the stack is [0, 1, 2, 4, 3, 5, 6]. We have the following code:

```

1 def dftIter(g, s):
2     '''not preserving the same discovery ordering'''
3     n = len(g)
4     orders = []
5     colors = [STATE.white] * n
6     stack = [s]
7
8     orders.append(s) # track gray order
9     colors[s] = STATE.gray
10
11    while stack:
12        u = stack.pop()
13
14        for v in g[u]:
15            if colors[v] == STATE.white:
16                colors[v] = STATE.gray
17                stack.append(v)

```

```

18     orders.append(v) # track gray order
19
20 return orders

```

Run the above code with `dftIter(al, 1)`, we have ordering $[[1, 2, 3, 4, 5, 6, 0],$ which is different from the recursive DFS version $[[1, 2, 0, 4, 3, 5, 6]]$. This is due to the different mechanism. To keep the same ordering of discovering order is not important. However, in our source code, we does provide a way to keep the same discovering ordering.

Method 2: Preserving both Discover and Finish Ordering Because in DFS each time, we start from u , we find one unvisited node u_1 and move forward to find one unvisited node u_2 of u_1 , until we met a node that has no unvisited adjacent nodes v . The visiting order will be u, u_1, u_2, \dots, v , For this process, we use a `stack` to save these nodes, each time to append it at the end and this marks the state as gray, and each time visit the end node in the stack. In this process, there is no pop out from the stack. If we are at v , which can not move the path farther, this marks the state as black, and means the state is complete and this node is ready to be moved out of the stack. Therefore, in the implementation, a bool variable `bAdj` to check if we are able to find an unvisited node or not. If we can not find one, then we pop out, if we can, we break the loop because we just need one unvisited node.

```

1 def dfsIter(g, s):
2     '''iterative dfs'''
3     v = len(g)
4     orders, complete_orders = [], []
5     colors = [STATE.white] * v
6     stack = [s]
7
8     orders.append(s) # track gray order
9     colors[s] = STATE.gray
10
11    while stack:
12        u = stack[-1]
13        bAdj = False
14        for v in g[u]:
15            if colors[v] == STATE.white:
16                colors[v] = STATE.gray
17                stack.append(v)
18                orders.append(v) # track gray order
19                bAdj = True
20                break
21
22        if not bAdj: # if no adjacent is found, pop out
23            # complete
24            colors[u] = STATE.black # this is not necessary in the
25            code, just to help track the state
26            complete_orders.append(u)

```

```

26     stack.pop()
27
28     return orders, complete_orders

```

Call `print(dfsIter(al, 0))`, the above code will have the same output as of the recursive implementation.

```
([0, 1, 2, 4, 3, 5, 6], [3, 6, 5, 4, 2, 1, 0])
```

Properties of DFS in Directed Graph

Applications

An animation of DFS is available <https://www.cs.usfca.edu/~galles/visualization/DFS.html>

11.4.3 Comparison of BFS and DFS

BFS and DFS is the most basic complete search in graph. They both search all vertices and edges by once, which made them share the same time complexity $O(|V| + |E|)$. We see, in the BFS, saving nodes of the gray state or black state has the same visiting ordering. Breadth-first search usually serves to find shortest path distances (and the associated predecessor subgraph) from a given source. Depth-first search is often a subroutine in another algorithm, as we shall see later in this chapter.

11.5 Discussion of Graph Search

As we will in the future chapters, basic BFS and DFS lays the fundations of all graph and tree-based search. Understanding the properties of graph search throughly in this chapter will ease our journey to explore more advanced graph algorithms. There are some properties related to graph that we need to learn before moving to the advanced algorithms.

Completeness In the context of search, a complete algorithm is one that guarantees that if a path to the goal exists, the algorithm will reach the goal. Note that *completeness* does not imply *optimality* of the found path.

For example, breadth-first search (BFS) is complete (and in fact optimal if step costs are identical at a given level), because it can find all paths starting from a given source vertex in the graph. (This might not be the case if step cost at a given level is not identical). while depth-first search (DFS) on trees is incomplete (consider infinite or repeated states).

12

Advanced Linear Data Structures-based Search

Array Search is to find a **sub-structure** on a given linear data structure(Chapter 7) or a virtual linear search space. Categorized by the definition of sub-structure:

- Define the sub-structure as a **particular item**: Usually the worst and average performance is $O(n)$. **Binary search** (Section 12.1) finds an item within an ordered data structure, each time, the search space is eliminated by half in size, which makes the worst time complexity $O(\log n)$. Using hashmap can gain us the best complexity of $O(1)$.
- Define the sub-structure as a **consecutive substructure** indexed by a start and end index (subarray) in the linear data structure, we introduce the **Sliding Window Algorithm** (Section 12.2). Compared with the brute force solution, it decrease the complexity from $O(n^2)$ to $O(n)$. If the sub-structure is **predefined pattern**, we need pattern matching algorithms. This usually exists in string data structure, and we do string pattern matching (Section ??).

12.1 Binary Search

To search in a sorted array or string using brute force with a for loop, it takes $O(n)$ time. Binary search is designed to reduce search time if the array or string is already sorted. It uses the divide and conquer method; each time we compare our target with the middle element of the array and with the comparison result to decide the next search region: either the left half or the right half. Therefore, each step we filter out half of the array which

gives the time complexity function $T(n) = T(n/2) + O(1)$, which decrease the time complexity to $O(\log n)$.

Binary Search can be applied to different tasks:

1. Find Exact target, find the first position that $\text{value} \geq \text{target}$, find the last position that $\text{value} \leq \text{target}$. (this is called lower_bound, and upper_bound).

12.1.1 Standard Binary Search and Python Module bisect

Binary search is usually carried out on a Static sorted array or 2D matrix. There are three basic cases: (1) find the exact target that $\text{value} = \text{target}$; If there are duplicates, we are more likely to be asked to (2) find the first position that has $\text{value} \geq \text{target}$; (3) find the first position that has $\text{value} \leq \text{target}$. Here, we use two example array: one without duplicates and the other has duplicates.

```
1 a = [2, 4, 5, 9]
2 b = [0, 1, 1, 1, 1]
```

Find the Exact Target This is the most basic application of binary search. We can set two pointers, l and r. Each time we compute the middle position, and check if it is equal to the target. If it is, return the position; if it is smaller than the target, move to the left half, otherwise, move to the right half. The Python code is given:

```
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
```

Now, run the example:

```
1 print("standard_binary_search: ", standard_binary_search(a,3),
      standard_binary_search(a,4), standard_binary_search(b, 1))
```

The print out is:

```
1 standard_binary_search: -1 1 2
```

From the example, we can see that multiple **duplicates** of the target exist, it can possibly return any one of them. And for the case when the target does not exist, it simply returns -1. In reality, we might need to find a position where we can potentially insert the target to keep the sorted array

sorted. There are two cases: (1) the first position that we can insert, which is the first position that has $\text{value} \geq \text{target}$ (2) and the last position we can insert, which is the first position that has $\text{value} > \text{target}$. For example, if we try to insert 3 in a, and 1 in b, the first position should be 1 and 1 in each array, and the last position is 1 and 6 instead. For these two cases, we have a Python built-in Module **bisect** which offers two methods: `bisect_left()` and `bisect_right()` for these two cases respectively.

Find the First Position that $\text{value} \geq \text{target}$ This way the target position separates the array into two halves: $\text{value} < \text{target}$, target_position , $\text{value} \geq \text{target}$. In order to meet the purpose, we make sure that if $\text{value} < \text{target}$, we move to the right side, else, move to the left side.

```

1 # bisect_left , no longer need to check the mid element ,
2 # it separate the list in to two halfs: value < target , mid ,
3     value >= target
4 def bisect_left_raw(lst , target):
5     l , r = 0 , len(lst)-1
6     while l <= r:
7         mid = l + (r-1)//2
8         if lst [mid] < target: # move to the right half if the
9             value < target , till
10            l = mid + 1 #[mid+1, right]
11        else:# move to the left half is value >= target
12            r = mid - 1 #[left , mid-1]
13    return l # the final position is where

```

Find the First Position that $\text{value} > \text{target}$ This way the target position separates the array into two halves: $\text{value} \leq \text{target}$, target_position , $\text{value} > \text{target}$. Therefore, we simply change the condition of if $\text{value} < \text{target}$ to if $\text{value} \leq \text{target}$, then we move to the right side.

```

1 #bisect_right: separate the list into two halfs: value<= target ,
2     mid , value > target
3 def bisect_right_raw(lst , target):
4     l , r = 0 , len(lst)-1
5     while l <= r:
6         mid = l + (r-1)//2
7         if lst [mid] <= target:
8             l = mid + 1
9         else:
10            r = mid -1
11    return l

```

Now, run an example:

```

1 print("bisect left raw: find 3 in a : ", bisect_left_raw(a,3) , ' '
2     find 1 in b: ', bisect_left_raw(b, 1))
2 print("bisect right raw: find 3 in a : ", bisect_right_raw(a, 3) ,
3     'find 1 in b: ', bisect_right_raw(b, 1))

```

The print out is:

```

1 bisect left raw: find 3 in a : 1 find 1 in b: 1
2 bisect right raw: find 3 in a : 1 find 1 in b: 6

```

Bonus For the last two cases, if we return the position as l-1, then we get the last position that value < target, and the last position value <= target.

Python Built-in Module bisect This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. It offers six methods as shown in Table 12.1. However, only two are most commonly used: bisect_left and bisect_right. Let's see some exam-

Table 12.1: Methods of **bisect**

Method	Description
bisect_left(a, x, lo=0, hi=len(a))	The parameters lo and hi may be used to specify a subset of the list; the function is the same as bisect_left_raw
bisect_right(a, x, lo=0, hi=len(a))	The parameters lo and hi may be used to specify a subset of the list; the function is the same as bisect_right_raw
bisect(a, x, lo=0, hi=len(a))	Similar to bisect_left(), but returns an insertion point which comes after (to the right of) any existing entries of x in a.
insort_left(a, x, lo=0, hi=len(a))	This is equivalent to a.insert(bisect.bisect_left(a, x, lo, hi), x).
insort_right(a, x, lo=0, hi=len(a))	This is equivalent to a.insert(bisect.bisect_right(a, x, lo, hi), x).
insort(a, x, lo=0, hi=len(a))	Similar to insert_left(), but inserting x in a after any existing entries of x.

plary code:

```

1 from bisect import bisect_left, bisect_right, bisect
2 print("bisect left: find 3 in a :", bisect_left(a,3), 'find 1 in'
      'b: ', bisect_left(b, 1)) # lower_bound, the first position
      # that value>= target
3 print("bisect right: find 3 in a :", bisect_right(a, 3), 'find 1'
      'in b: ', bisect_right(b, 1)) # upper_bound, the last
      # position that value <= target

```

The print out is:

```

1 bisect left: find 3 in a : 1 find 1 in b: 1
2 bisect right: find 3 in a : 1 find 1 in b: 6

```

12.1.2 Binary Search in Rotated Sorted Array

The extension of the standard binary search is on array that the array is ordered in its own way like rotated array.

Binary Search in Rotated Sorted Array (See LeetCode problem, 33. Search in Rotated Sorted Array (medium). Suppose an array (without duplicates) sorted in ascending order is rotated at some pivot unknown to you beforehand. (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2). You are given a target value to search. If found in the array return its index, otherwise return -1. You may assume no duplicate exists in the array.

Example 1:

```
Input: nums = [3, 4, 5, 6, 7, 0, 1, 2], target = 0
Output: 5
```

Example 2:

```
Input: nums = [4, 5, 6, 7, 0, 1, 2], target = 3
Output: -1
```

In the rotated sorted array, the array is not purely monotonic. Instead, there is one drop in the array because of the rotation, where it cuts the array into two parts. Suppose we are starting with a standard binary search with example 1, at first, we will check index 3, then we need to move to the right side? Assuming we compare our middle item with the left item,

```
if nums[mid] > nums[1]: # the left half is sorted
elif nums[mid] < nums[1]: # the right half is sorted
else: # for case like [1,3], move to the right half
```

For a standard binary search, we simply need to compare the target with the middle item to decide which way to go. In this case, we can use objection. Check which side is sorted, because no matter where the left, right and the middle index is, there is always one side that is sorted. So if the left side is sorted, and the value is in the range of the [left, mid], then we move to the left part, else we object the left side, and move to the right side instead.

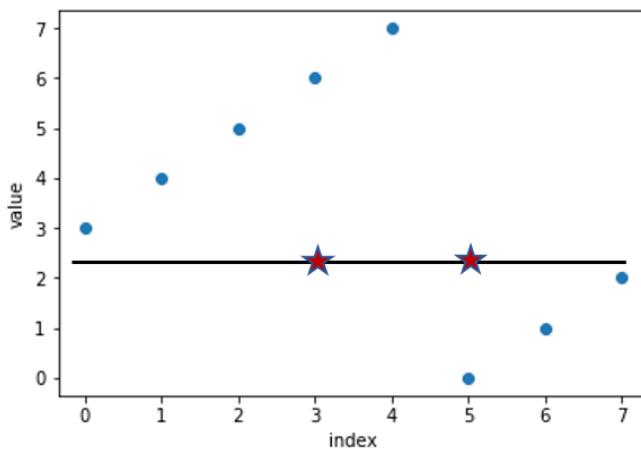


Figure 12.1: Example of Rotated Sorted Array

The code is shown:

```

1 '''implemente the rotated binary search'''
2 def RotatedBinarySearch(nums, target):
3     if not nums:
4         return -1
5
6     l, r = 0, len(nums)-1
7     while l<=r:
8         mid = l+ (r-1)//2
9         if nums[mid] == target:
10            return mid
11         if nums[1] < nums[mid]: # if the left part is sorted
12             if nums[1] <= target <= nums[mid]:
13                 r = mid-1
14             else:
15                 l = mid+1
16         elif nums[1] > nums[mid]: # if the right side is
17             if nums[mid] <= target <= nums[r]:
18                 l = mid+1
19             else:
20                 r = mid-1
21         else:
22             l = mid + 1
23     return -1

```

 What happens if there is duplicates in the rotated sorted array?

In fact, similar comparing rule applies:

```

if nums[mid] > nums[1]: # the left half is sorted
elif nums[mid] < nums[1]: # the right half is sorted
else: # for case like [1,3], or [1, 3, 1, 1, 1] or [3, 1, 2,
      3, 3, 3]
    only l++

```

12.1.3 Binary Search on Result Space

If the question gives us the context: the target is in the range [left, right], we need to search the first or last position that satisfy a condition function. We can apply the concept of standard binary search and bisect_left and bisect_right and its mutant. Where we use the condition function to replace the value comparison between target and element at middle position. The steps we need:

1. get the result search range [l, r] which is the initial value for l and r pointers.

2. decide the valid function to replace such as if $\text{lst}[\text{mid}] < \text{target}$
3. decide which binary search we use: standard, `bisect_left`/ `bisect_right` or its mutant.

For example:

12.1 441. Arranging Coins (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: Given a number $n \geq 1$, the minimum row is 1, and the maximum is n . Therefore, our possible result range is $[1, n]$. These can be treated as indexes of the sorted array. For a given row, we write a function to check if it is possible. We need a function $r*(r+1)//2 \leq n$. For this problem, we need to search in the range of $[1, n]$ to find the last position that is valid. This is `bisect_left` or `bisect_right`, where we use the function replace the condition check:

```
1 def arrangeCoins(self, n):
2     def isValid(row):
3         return (row*(row+1))//2 <= n
4     # we need to find the last position that is valid (<=)
5     def bisect_right():
6         l, r = 1, n
7         while l <= r:
8             mid = l + (r-1) // 2
9             if isValid(mid): # replaced compared with the
10                standard binary search
11                l = mid + 1
12            else:
13                r = mid - 1
14        return l-1
15    return bisect_right()
```

12.2 278. First Bad Version. 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.

Solution: we keep doing binary search until we have searched all possible areas.

```

1 class Solution(object):
2     def firstBadVersion(self, n):
3         """
4             :type n: int
5             :rtype: int
6         """
7         l, r=0,n-1
8         last = -1
9         while l<=r:
10             mid = l+(r-1)//2
11             if isBadVersion(mid+1): #move to the left , mid
12                 is index , s
13                 r=mid-1
14                 last = mid+1 #to track the last bad one
15             else:
16                 l=mid-1
17         return last

```

12.1.4 LeetCode Problems

12.1 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 traget
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:

$$\begin{bmatrix} 1, & 3, & 5, & 7 \\ 10, & 11, & 16, & 20 \\ 23, & 30, & 34, & 50 \end{bmatrix}$$

Given target = 3, return true.

Solution: 2D matrix search, time complexity from $O(n^2)$ to $O(lgm + lgn)$.

```
1 def searchMatrix(self , matrix , target):
2     """
3         :type matrix: List[ List[ int ] ]
4         :type target: int
5         :rtype: bool
6     """
7
8     if not matrix:
9         return False
10    row , col = len(matrix) , len(matrix[0])
11    if row==0 or col==0: #for []
12        return False
13    sr , er = 0 , row-1
14    #first search the mid row
15    while sr<=er:
16        mid = sr+(er-sr)//2
17        if target>matrix[mid][-1]: #go to the right
18            side
19                sr=mid+1
20            elif target < matrix[mid][0]: #go to the left
21                side
22                    er = mid-1
23            else: #value might be in this row
24                #search in this row
25                lc , rc = 0 , col-1
26                while lc<=rc:
27                    midc = lc+(rc-lc)//2
28                    if matrix[mid][midc]==target:
29                        return True
30                    elif target<matrix[mid][midc]: #go to
31                        left
32                            rc=midc-1
33                        else:
34                            lc=midc+1
35                        return False
36                return False
37
38    return False
```

Also, we can treat it 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)

12.2 Two-pointer Search

There are actually 50/900 problems on LeetCode are tagged as two pointers. Two pointer search algorithm are normally used to refer to searching that use two pointer in one for/while loop over the given data structure. Therefore,

this part of algorithm gives linear performance as of $O(n)$. While, it does not refer to situation such as searching a pair of items in an array that sums up to a given target value, then two nested for loops are needed to search all the possible pairs. There are different ways to put these two pointers:

1. Equi-directional: Both start from the beginning; we have **slow-faster pointer**, **sliding window algorithm**.
 2. Opposite-directional: One at the start and the other at the end, they move close to each other and meet in the middle, (->-<-).

In order to use two pointers, most times the data structure needs to be ordered in some way, and decrease the time complexity from $O(n^2)$ or $O(n^3)$ of two/three nested for/while loops to $O(n)$ of just one loop with two pointers and search each item just one time. In some cases, the time complexity is highly dependable on the data and the criteria we set.

As shown in Fig. 12.2, the pointer i and j can decide: a pair or a subarray (with all elements starts from i and end at j). We can either do search related with a pair or a subarray. For the case of subarray, the algorithm is called sliding window algorithm. As we can see, two pointers and sliding window algorithm can be used to solve K sum (Section ??), most of the subarray (Section ??), and string pattern match problems (Section ??).

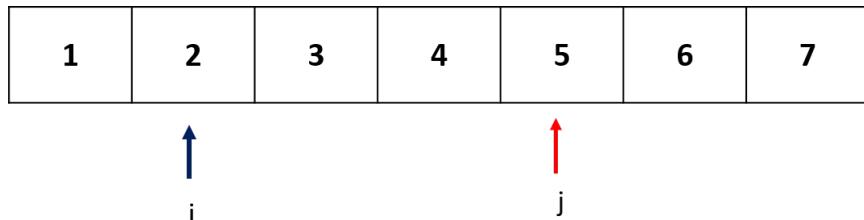


Figure 12.2: Two pointer Example

Two pointer algorithm is less of a talk and more of problem attached. We will explain this type of algorithm in virtue of both the leetcode problems and definition of algorihtms. To understand two pointers techniques, better to use examples, here we use two examples: use slow-faster pointer to find the median and Floyd's fast-slow pointer algorithm for loop detection in an array/linked list and two pointers to get two sum.

12.2.1 Slow-fast Pointer

Find middle node of linked list The simplest example of slow-fast pointer application is to get the middle node of a given linked list. (LeetCode problem: 876. Middle of the Linked List)

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])

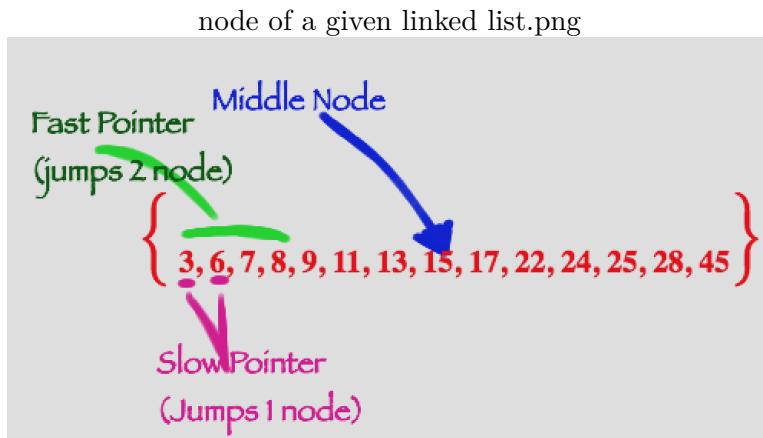


Figure 12.3: Slow-fast pointer to find middle

We place two pointers simultaneously at the head node, each one moves at different paces, the slow pointer moves one step and the fast moves two steps instead. When the fast pointer reached the end, the slow pointer will stop at the middle. For the loop, we only need to check on the faster pointer, make sure fast pointer and fast.next is not None, so that we can successfully visit the fast.next.next. When the length is odd, fast pointer will point at the end node, because fast.next is None, when its even, fast pointer will point at None node, it terminates because fast is None.

```

1 def middleNode(self, head):
2     slow, fast = head, 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) Given a linked list which has a cycle, as shown in Fig. 12.4. To check the existence of the cycle is quite simple. We do exactly the same as traveling by the slow and fast pointer above, each at one and two steps. (LeetCode Problem: 141. Linked List Cycle). The code is pretty much the same with the only

difference been that after we change the fast and slow pointer, we check if they are the same node. If true, a cycle is detected, else not.

```

1 def hasCycle(self, 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

```

In order to know the starting node of the cycle. Here, we set the distance of 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.

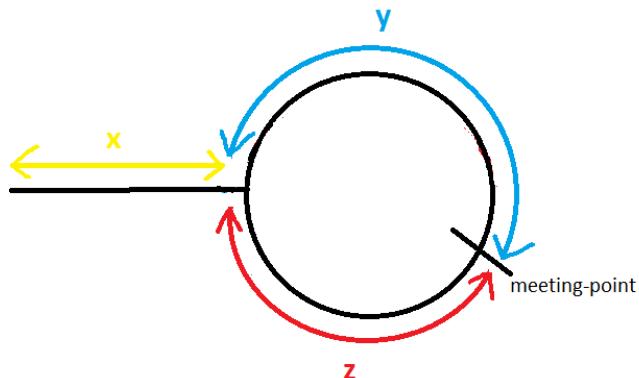


Figure 12.4: Floyd's Cycle finding Algorithm

Now, let's try to device the algorithm. Both slow and fast pointer starts at position 0, the node index they travel each step is: $[0,1,2,3,\dots,k]$ and $[0,2,4,6,\dots,2k]$ for slow and fast pointer respectively. Therefore, the total distance travelled by the slow pointer is half of the distance travelled by the fat pointer. From the above figure, we have the distance travelled by slow pointer to be $d_s = x+y$, and for the fast pointer $d_f = x+y+z+y = x+2y+z$. With the relation $2 * d_s = d_f$. We will eventually get $x = z$. Therefore, by moving slow pointer to the start of the linked list after the meeting point, and making both slow and fast pointer to move one node at a time, they will meet at the starting node of the cycle. (LeetCode problem: 142. Linked List Cycle II (medium)).

```

1 def detectCycle(self, head):
2     slow = fast = head
3     bCycle = False
4     while fast and fast.next:

```

```

5     slow = slow.next
6     fast = fast.next.next
7     if slow == fast: # a cycle is found
8         bCycle = True
9         break
10
11    if not bCycle:
12        return None
13    # reset the slow pointer to find the starting node
14    slow = head
15    while fast and slow != fast:
16        slow = slow.next
17        fast = fast.next
18    return slow

```

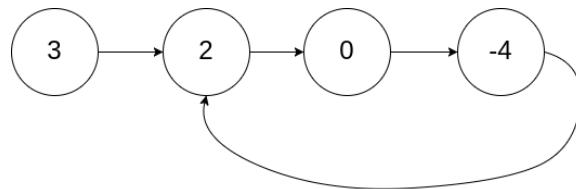


Figure 12.5: One example to remove cycle

In order to remove the cycle as shown in Fig. 12.5, the starting node is when slow and fast intersect, the last fast node before they meet. For the example, we need to set -4 node's next node to None. Therefore, we modify the above code to stop at the last fast node instead:

```

1  # reset the slow pointer to find the starting node
2  slow = head
3  while fast and slow.next != fast.next:
4      slow = slow.next
5      fast = fast.next
6  fast.next = None

```

12.2.2 Opposite-directional Two pointer

Two pointer is usually used for searching a pair in the array. There are cases the data is organized in a way that we can search all the result space by placing two pointers each at the start and rear of the array and move them to each other and eventually meet and terminate the search process. The search target should help us decide which pointer to move at that step. This way, each item in the array is guaranteed to be visited at most one time by one of the two pointers, thus making the time complexity to be $O(n)$. Binary search used the technique of two pointers too, the left and right pointer together decides the current searching space, but it erase of half searching space at each step instead.

Two Sum - Input array is sorted 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. The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. (LeetCode problem: 167. Two Sum II - Input array is sorted (easy).)

```
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore index1 = 1,
index2 = 2.
```

Due to the fact that the array is sorted which means in the array $[s, s+e, \dots, e]$, the sum of any two integer is in range of $[s+s, e+e]$. By placing two pointers each start from s and e , we started the search space from the middle of the possible range. $[s+s, s+e, e+e]$. Compare the target t with the sum of the two pointers v_1 and v_2 :

1. $t == v_1 + v_2$: found
2. $v_1 + v_2 < t$: we need to move to the right side of the space, then we increase v_1 to get larger value.
3. $v_1 + v_2 > t$: we need to move to the left side of the space, then we decrease v_2 to get smaller value.

```
1 def twoSum( self , numbers , target ):
2     #use two pointers
3     n = len(numbers)
4     i , j = 0 , n-1
5     while i < j:
6         temp = numbers[ i ] + numbers[ j ]
7         if temp == target:
8             return [ i+1, j+1]
9         elif temp < target:
10            i += 1
11        else:
12            j -= 1
13    return []
```

12.2.3 Sliding Window Algorithm

Given an array, imagine that we have a fixed size window as shown in Fig. 12.6, and we can slide it forward each time. If we are asked to compute the sum of each window, the bruteforce solution would be $O(kn)$ 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. A sliding window algorithm applied here used the property that the sum of the current window (S_c) can be computed from the last window (S_l) knowing the items



Figure 12.6: Sliding Window Algorithm

that just slided out and moved in as a_o and a_i . Then $S_c = S_l - a_o + a_i$. Not necessarily using sum, we generalize it as state, if we can compute S_c from S_l , a_o and a_i in $O(1)$, a function $S_c = f(S_l, a_o, a_i)$ then we name this **sliding window property**. Therefore the time complexity will be decreased to $O(n)$.

```

1 def fixedSlideWindow(A, k):
2     n = len(A)
3     if k >= n:
4         return sum(A)
5     # compute the first window
6     acc = sum(A[:k])
7     ans = acc
8     # slide the window
9     for i in range(n-k): # i is the start point of the window
10        j = i + k # j is the end point of the window
11        acc = acc - A[i] + A[j]
12        ans = max(ans, acc)
13    return ans

```

When to use sliding window It is important to know when we can use sliding window algorithm, we summarize three important standards:

1. It is a subarray/substring problem.
2. **sliding window property:** The requirement of the sliding window satisfy the sliding window property.
3. **Completeness:** by moving the left and right pointer of the sliding window in a way that we can cover all the search space. Sliding window algorithm is about optimization problem, and by moving the left and right pointer we can search the whole searching space. **Therefore, to testify that if applying the sliding window can cover the whole search space and guarantee the completeness decide if the method works.**

For example, 644. Maximum Average Subarray II (hard) does not satisfy the completeness. Because the average of subarray does not follow a certain order that we can decided how to move the window.

Flexible Sliding Window Algorithm Another form of sliding window where the window size is flexible, and it can be used to solve a lot of real problems related to subarray or substring that is conditioned on some pattern. Compared with the fixed size window, we can first fix the left pointer, and push the right pointer to enlarge the window in order to find a subarray satisfy a condition. Once the condition is met, we save the optimal result and shrink the window by moving the left pointer in a way that we can set up a new starting pointer to the window (shrink the window). At any point in time only one of these pointers move and the other one remains fixed.

Sliding Window Algorithm with Sum In this part, we list two examples that we use flexible sliding window algorithms to solve subarray problem with sum condition.

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. (LeetCode Problem: 209. Minimum Size Subarray Sum (medium)).

Example :

Input: $s = 7$, $\text{nums} = [2, 3, 1, 2, 4, 3]$

Output: 2

Explanation: the subarray $[4, 3]$ has the minimal length under the problem constraint.

As we have shown in Fig. 12.7, the prefix sum is the subarray starts with the first item in the array, we know that the sum of the subarray is monotonically increasing as the size of the subarray increase. Therefore, we place a 'window' with left and right as i and j at the first item first. The steps are as follows:

1. Get the optimal subarray starts from current i , 0: Then we first move the j pointer to include enough items that $\text{sum}[0:j+1] \geq s$, this is the process of getting the optimial subarray that starts with 0. And assume j stops at e_0
2. Get the optimal subarray ends with current j , e_0 : we shrink the window size by moving the i pointer forward so that we can get the optimal subarray that ends with current j and the optimal subarray starts from s_0 .
3. Now, we find the optimal solution for subproblem $[0:i, 0:j]$ (the start point in range $[0, i]$ and end point in range $[0,j]$. Starts from next i and j , and repeat step 1 and 2.

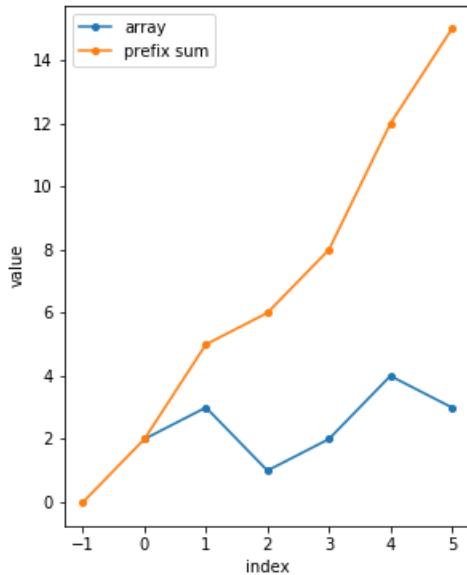


Figure 12.7: The array and the prefix sum

The above process is a standard flexible window size algorithm, and it is a complete search which searched all the possible result space. Both j and i pointer moves at most n , it makes the total operations to be at most $2n$, which we get time complexity as $O(n)$.

```

1 def minSubArrayLen(self, s, nums):
2     ans = float('inf')
3     n = len(nums)
4     i = j = 0
5     acc = 0 # acc is the state
6     while j < n:
7         acc += nums[j]# increase the window size
8         while acc >= s:# shrink the window to get the optimal
9             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

```



What happens if there exists negative number in the array?

Sliding window algorithm will not work any more, because the sum of the subarray is no longer monotonically increase as the size increase. 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 problem: 325. Maximum Size Subarray Sum Equals k,
 325. Maximum Size Subarray Sum Equals k (hard))

More similar problems:

1. 674. Longest Continuous Increasing Subsequence (easy)

Sliding Window Algorithm with Substring For substring problems, to be able to use sliding window, $s[i,j]$ should be gained from $s[i,j-1]$ and $s[i-1,j-1]$ should be gained from $s[i,j-1]$. Given a string, find the length of the longest substring without repeating characters. (LeetCode Problem: 3. Longest Substring Without Repeating Characters (medium))

Example 1:

```
Input: "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: "bbbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

First, we know it is a substring problem. Second, it asks to find substring that only has unique chars, we can use hashmap to record the chars in current window, and this satisfy the sliding window property. When the current window violates the condition (a repeating char), we shrink the window in a way to get rid of this char in the current window by moving the i pointer one step after this char.

```
1 def lengthOfLongestSubstring(self, s):
2     if not s:
3         return 0
4     n = len(s)
5     state = set()
6     i = j = 0
7     ans = -float('inf')
8     while j < n:
9         if s[j] not in state:
10             state.add(s[j])
11             ans = max(ans, j-i)
12         else:
13             # shrink the window: get this char out of the window
14             while s[i] != s[j]: # find the char
15                 state.remove(s[i])
16                 i += 1
17             # skip this char
18             i += 1
19             j += 1
20     return ans if ans != -float('inf') else 0
```

Now, let us see another example with string and given a pattern to match. Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n). (LeetCode Problem: 76. Minimum Window Substring (hard))

Example :

Input: S = "ADOBECODEBANC" , T = "ABC"
Output: "BANC"

In this problem, the desirable window is one that has all characters from T. The solution is pretty intuitive. We keep expanding the window by moving the right pointer. When the window has all the desired characters, we contract (if possible) and save the smallest window till now. The only difference compared with the above problem is the definition of desirable: we need to compare the state of current window with the required state in T. They can be handled as a hashmap with character as key and frequency of characters as value.

```

1 def minWindow(self, s, t):
2     dict_t = Counter(t)
3     state = Counter()
4     required = len(dict_t)
5
6     # left and right pointer
7     i, j = 0, 0
8
9     formed = 0
10    ans = float("inf"), None # min len, and start pos
11
12    while j < len(s):
13        char = s[j]
14        # record current state
15        if char in dict_t:
16            state[char] += 1
17            if state[char] == dict_t[char]:
18                formed += 1
19
20        # Try and contract the window till the point where it
21        # ceases to be 'desirable'.
22        # bPrint = False
23        while i <= j and formed == required:
24            # if not bPrint:
25            #     print('found:', s[i:j+1], i, j)
26            #     bPrint = True
27            char = s[i]
28            if j-i+1 < ans[0]:
29                ans = j - i + 1, i
30            # change the state
31            if char in dict_t:
32                state[char] -= 1
33                if state[char] == dict_t[char]-1:
34                    formed -= 1

```

```

34         # Move the left pointer ahead,
35         i += 1
36
37         # Keep expanding the window
38         j += 1
39         # if bPrint:
40         #     print('move to:', s[i:j+1], i, j)
41     return "" if ans[0] == float("inf") else s[ans[1] : ans[1] +
42         ans[0]]

```

The process would be:

```

found: ADOBEC 0 5
move to: DOBECO 1 6
found: DOBECODEBA 1 10
move to: ODEBAN 6 11
found: ODEBANC 6 12
move to: ANC 10 13

```

Three Pointers and Sliding Window Algorithm Sometimes, by manipulating two pointers are not enough for us to get the final solution.

12.1 930. Binary Subarrays With Sum. 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 bolded below:
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]

```

Note: A.length <= 30000, 0 <= S <= A.length, A[i] is either 0 or 1.

For example in the following problem, if we want to use two pointers to solve the problem, we would find we miss the 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_h , 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, it is also a mutant sliding window algorithm.

```

1 class Solution:
2     def numSubarraysWithSum(self, A, S):
3         i_lo, i_hi, j = 0, 0, 0 #i_lo <= j
4         sum_window = 0

```

```

5     ans = 0
6     while j < len(A):
7
8         sum_window += A[j]
9
10        while i_lo < j and sum_window > S:
11            sum_window -= A[i_lo]
12            i_lo += 1
13            # up till here, it is standard sliding window
14
15            # now set the extra pointer at the same
16            # location of the i_lo
17            i_hi = i_lo
18            while i_hi < j and sum_window == S and not A[
19                i_hi]:
20                i_hi += 1
21            if sum_window == S:
22                ans += i_hi - i_lo + 1
23
24            j += 1 #increase the pointer at last so that we
25            do not need to check if j<len again
26
27        return ans

```

Summary Sliding Window is a powerful tool for solving certain subarray/substring related problems. The normal situations where we use sliding window is summarized:

- Subarray: for an array with numerical value, it requires all positive/negative values so that the prefix sum/product has monotonicity.
- Substring: for an array with char as value, it requires the state of each subarray does not relate to the order of the characters (anagram-like state) so that we can have the sliding window property.

The steps of using sliding windows:

1. Initialize the left and right pointer;
2. Handle the right pointer and record the state of the current window;
3. While the window is in the state of desirable: record the optimal solution, move the left pointer and record the state (change or stay unchanged).
4. Up till here, the state is not desirable. Move the right pointer in order to find a desirable window;

12.2.4 LeetCode Problems

Sliding Window

12.1 76. Minimum Window Substring

12.2 438. Find All Anagrams in a String

12.3 30. Substring with Concatenation of All Words

12.4 159. Longest Substring with At Most Two Distinct Characters

12.5 567. Permutation in String

12.6 340. Longest Substring with At Most K Distinct Characters

12.7 424. Longest Repeating Character Replacement

13

Non-linear Recursive Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, that *incrementally* builds candidates to the solutions. It is able to enumerate all possible states and guarantee that we would be able to find the “correct” solution. Moreover, along the way, as soon as it determines that a candidate cannot possibly lead to a valid *complete solution*, it abandons this *partial candidate* and “backtracks” (return to the upper level) and reset to the upper level’s state so that the search process can continue to explore the next branch to provide more efficiency.

We can model the combinatorial search solution as a vector $s = (s_0, s_1, \dots, s_{n-1})$, where each s_i is selected from a finite ordered set A . Such a vector might represent an arrangement where s_i contains the i -th element of the permutation. Or 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. At each step in the backtracking algorithm, we try to extend the last partial solution $s = (s_0, s_1, \dots, s_k)$ by adding another event at the end. And then we testify our partial solution with the desired solution to decide to (1) either collect this partial solution; and or (2) adding s_{k+1} ; or (3) backtrack and reset to previous state and go to next branch.

Visualize Backtracking Let us represent different s as a node in a tree. Initial state $s = []$ as the root node. The first level represents all possible states for $s = (s_0)$ of length 1, and the second level for $s = (s_0, s_1)$ of length 2. And the edge represents making a choice out of all items in the ordered set A . If we reach to end condition (leaf candidates) we succeed and the search stop. If the partial candidate can not satisfy the constraint, we return to the root node, and reset the state (‘backtrack’). The process is shown in

Fig. 13.1.

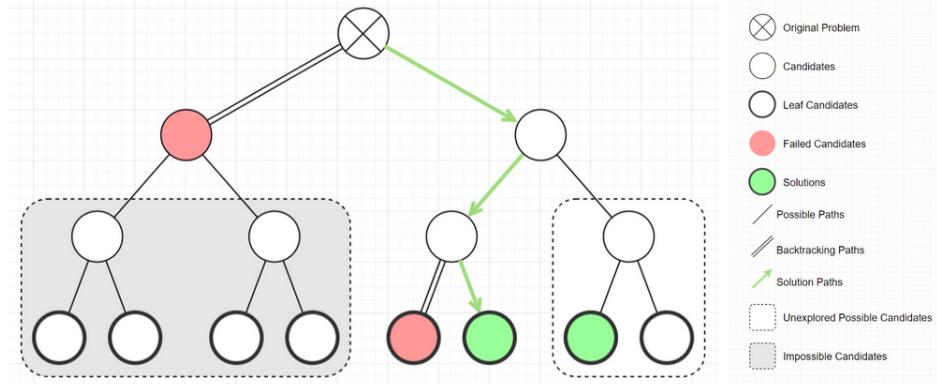


Figure 13.1: Tree of possibilities for a typical backtracking algorithm

Backtrack Template We list the template here which we summarized after viewing different backtracking algorithms. It usually is composed of two parts: initialization and main dfs backtracking. After we figure out our state vector s where in our template is `state_tracker` with total search tree depth n . In the main backtracking state: we first generate candidates according to previous states and then try out each candidate by iteration. We set the state before call recursive function to the next depth and reset the state after we return and move on to try next candidate.

```

1 def backtrack():
2     # initialization
3     A #a working data structure, either a list of candidates or a
4     # graph or a matrix representing a board
5     state_tracker = []*n
6     assist_state_tracker
7     # main backtracking
8     def dfs(d, n):
9         '''d: depth representing level in the tree'''
10        if d == n:
11            return
12        candidates = generate_candidates(state_tracker,
13                                         assist_state_tracker)
14        for c in candidates:
15            set_state(state_tracker, assist_state_tracker, c)
16            dfs(d+1, n)
17            reset_state(state_tracker, assist_state_tracker, c)
18
19    dfs(0, n)

```

Complexity Analysis The time complexity of backtracking can be obtained from analyzing the search tree. The worst case incurs when the

complete result occurs at the right most of the search tree, thus we need to traverse all the paths resulting visiting each node twice—one forward and one backward. Assume the cost to generate and visit a node is $O(2)$, and the total time complexity will be $O(|V|)$, where $|V|$ is the total nodes in the traverse tree.

Backtracking is all about choices and consequences, this is why backtracking is the most common algorithm for solving *constraint satisfaction problem (CSP)*¹, where the goal is to find a set of value assignments to certain variables that will satisfy specific mathematical equations and inequations. For example, Eight Queens puzzle, Map Coloring problem, Sudoku, Crosswords, and many other logic puzzles.

Properties and Applications To generalize the characters of backtracking:

1. **No Repetition and Completion:** It is a systematic generating method that avoids repetitions and missing any possible right solution. This property makes it ideal for solving combinatorial problems such as combination and permutation which requires us to enumerate all possible solutions.
2. **Search Pruning:** Because the final solution is built incrementally, in the process of working with partial solutions, we can evaluate the partial solution and prune branches that would never lead to the acceptable complete solution: either it is invalid configuration, or it is worse than known possible complete solution. With search pruning and we end up amortizely visiting each vertex less than once which is more efficient compared with an exhaustive graph search such as DFS and BFS.

In this chapter, the organization is as follows:

1. Show Property 1: We will first show how backtrack construct the complete solution incrementally and how it backtracks to its previous state in Sec. 13.1.
 - (a) **On Implicit Graph:** start with the combination and permutation problem to show us how the backtracking works in Section 13.1.1 and 13.1.2 with simple and commonly seen combinatorial problems: combination and permutation.
 - (b) **On Explicit Graph:** Enumerating all paths between the source and target vertex in a graph drawing in Section 13.1.3. Similarly, it can be applied on enumerate all spanning trees, graph partition.

¹CSPs are mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations, visit https://en.wikipedia.org/wiki/Constraint_satisfaction_problem for more information

2. Show Property 2: we demonstrate the application of search pruning in backtracking through CSP problems in Section 13.2.

13.1 Enumeration

13.1.1 Permutation

Before I throw you more theoretical talking, let us look at an example: Given a set of integers 1, 2, 3, enumerate all possible permutations using all items from the set without repetition. A permutation describes an arrangement or ordering of items. It is trivial to figure out that we can have the following six permutations: [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], and [3, 2, 1].

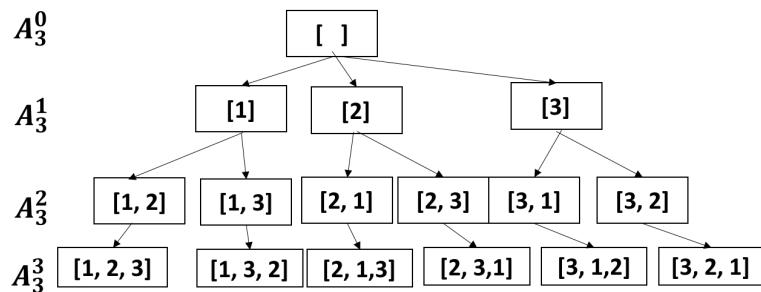


Figure 13.2: The state transfer graph of Permutation

This is a typical combinatorial problem, the process of generating all valid permutations is visualized in Fig. 13.2. To construct the final solution, we can start from an empty ordering shown at the first level, $[]$. Then we try to add one item where we have three choices :1, 2, and 3. We get three partial solutions $[1]$, $[2]$, $[3]$ at the second level. Next, for each of these partial solutions, we have two choices, for $[1]$, we can either put 2 or 3 first. Similarly, for $[2]$, we can do either 1 and 3, and so on. Given n distinct items, the number of possible permutations are $n * (n - 1) * \dots * 1 = n!$.

Implicit Graph In the graph, each node is either a partial or final solution. If we look it as a tree, the internal node is a partial solution and all leaves are final solutions. One edge represents generating the next solution based on the current solution. The vertices and edges are not given by an explicitly defined graph or trees, the vertices are generated on the fly and the edges are implicit relation between these nodes.

Backtracking VS DFS The implementation of the state transfer we can use either BFS or DFS on the implicit vertices. DFS is preferred because theoretically it took $O(\log n!)$ space used by stack, while if use BFS, the number of vertices saved in the queue can be close to $n!$. With recursive

DFS, we can start from node $[]$, and traverse to $[1,2]$, then $[1,2,3]$. Then we backtrack to $[1,2]$, backtrack to $[1]$, and go to $[1, 3]$, to $[1, 3, 2]$. To clear the relation between backtracking and DFS, we can say backtracking is a complete search technique which systematically builds the search tree (implicitly and not graph) and DFS is an ideal way to implement it.

Back to Permutation We can generalize Permutation, Permutations refer to the permutation of n things taken k at a time without repetition, the math formula is $A_n^k = n * (n - 1) * (n - 2) * \dots * k$. In Fig. 13.2, we can see from each level k shows all the solution of A_n^k . The generation of A_n^k is shown in the following Python Code.

```

1 def A_n_k(a, n, k, depth, used, curr, ans):
2     """
3         Implement permutation of k items out of n items
4         depth: start from 0, and represent the depth of the search
5         used: track what items are in the partial solution from the
6             set of n
7         curr: the current partial solution
8         ans: collect all the valide solutions
9     """
10    if depth == k: #end condition
11        ans.append(curr[:])
12        return
13    for i in range(n):
14        if not used[i]:
15            # generate the next solution from curr
16            curr.append(a[i])
17            used[i] = True
18            # move to the next solution
19            A_n_k(a, n, k, depth+1, used, curr, ans)
20
21            #backtrack to previous partial state
22            curr.pop()
23            used[i] = False
24    return

```

Give the input of $a=[1, 2, 3]$, we call the above function with the following code:

```

1 a = [1, 2, 3]
2 n = len(a)
3 ans = [[None]]
4 used = [False] * len(a)
5 ans = []
6 A_n_k(a, n, n, 0, used, [], ans)
7 print(ans)

```

The output is:

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

```

1 [1]
2 [1, 2]
3 [1, 2, 3]
4 backtrack: [1, 2]
5 backtrack: [1]
6 [1, 3]
7 [1, 3, 2]
8 backtrack: [1, 3]
9 backtrack: [1]
10 backtrack: []
11 [2]
12 [2, 1]
13 [2, 1, 3]
14 backtrack: [2, 1]
15 backtrack: [2]
16 [2, 3]
17 [2, 3, 1]
18 backtrack: [2, 3]
19 backtrack: [2]
20 backtrack: []
21 [3]
22 [3, 1]
23 [3, 1, 2]
24 backtrack: [3, 1]
25 backtrack: [3]
26 [3, 2]
27 [3, 2, 1]
28 backtrack: [3, 2]
29 backtrack: [3]
30 backtrack: []

```

To notice, in line 10 of `A_n_k` we used deep copy `curr[::]`. Because if not, we need up getting all empty list due to the fact that `curr` is used to track all vertices, when it backtrack to the root node, it eventually become `[]`.

Two Passes Therefore, we can say backtrack visits these implicit vertices in two passes: First, the forward pass to build the solution incrementally. Second, the backward pass to backtrack to previous state. We can see within these two passes, the `curr` list is used as all vertices in the search tree, and it start with `[]` and end with `[]`. This is the core character of backtracking.

Time Complexity of Permutation In the example of permutation, we can see that backtracking only visit each state once. The complexity of this is similar to the graph traversal of $O(|V| + |E|)$, where $|V| = \sum_{i=0}^n A_n^k$, because it is a tree structure, $|E| = |v| - 1$. This actually makes the permutation problem NP-hard.

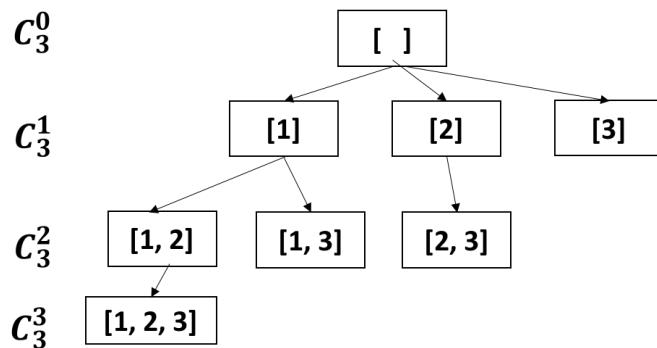


Figure 13.3: The state transfer graph of Combination

13.1.2 Combination

Continue from the permutation, combination refers to the combination of n things taken k at a time without repetition, the math formula C_n^k . Combination does not care about the ordering between chosen elements, such that $[a, b]$ and $[b, a]$ are considered as the same solution, thus only one of them can exist. Some useful formulation with combinations include $C_n^k = C_n^{n-k}$.

Here, we will demonstrate how we can use backtracking to implement an algorithm that just can traverse all the states and generate the power set (all possible subsets). We can see at each level, for each parent node, we would go through all of the elements in the array that has not been used before. For example, for $[]$, we get $[1], [2], [3]$; for $[1]$, we need $[2], [3]$ to get $[1, 2], [1, 3]$. Thus we use an index in the designed function to denote the start position for the elements to be combined. Also, we use DFS, which means the path is $[]->[1]->[1, 2]->[1, 2, 3]$, we would use recursive function because it is easier to implement and also we can spare us from using a stack to save these nodes, which can be long and it would not really bring the benefit of using iterative implementation. The key point here is after the recursive function returns to the last level, say after $[1, 2, 3]$ is generated, we would return to the previous state (this is why it is called backtrack!! Incremental and Backtrack).

Implementation The process is the similar to the implementation of permutation, except that we have one different variable, `start`. `start` is used to track the start index of the next candidate instead of use the `used` array to track the state of each item in the `curr` solution. `curr.pop()` is the soul for showing it is a backtracking algorithm!

```

1 def C_n_k(a, n, k, start, depth, curr, ans):
2     """
3         Implement combination of k items out of n items
4         start: the start of candidate
5         depth: start from 0, and represent the depth of the search
  
```

```

6 curr: the current partial solution
7 ans: collect all the valide solutions
8 !!!
9 if depth == k: #end condition
10    ans.append(curr[:])
11    return
12
13 for i in range(start, n):
14    # generate the next solution from curr
15    curr.append(a[i])
16    # move to the next solution
17    C_n_k(a, n, k, i+1, depth+1, curr, ans)
18
19    #backtrack to previous partial state
20    curr.pop()
21

```

Similarly, we call the following code:

```

1 a = [1, 2, 3]
2 n = len(a)
3 ans = [[None]]
4 ans = []
5 C_n_k(a, n, 2, 0, 0, [], ans)
6 print(ans)

```

The output is:

```
[[1, 2], [1, 3], [2, 3]]
```



Try to modify the above code so that you can collect the power set.

Note: To generate the power set, backtracking is NOT the only solution, if you are interested right now, check out Section 26.3.1.

Time Complexity of Combination Because backtracking ensures efficiency by visiting each state no more than once. For the combination(subset) problem, the total nodes of the implicit search graph/tree is $\sum_{k=0}^n C_n^k = 2^n$. We can look it as another way, there are in total n objects, and each object we can make two decisions: inside of the subset or not, therefore, this makes 2^n .

13.1.3 All Paths

Backtracking technique can be naturally used in graph path traversal. One example is to find all possible paths from a source to the target. One simpler occasion is when the graph has no cycles. Backtrack technique can enumerate all paths in the graph exactly once for each.

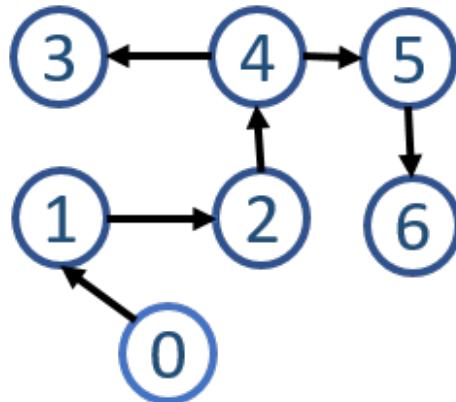


Figure 13.4: All paths from 0, include $0 \rightarrow 1$, $0 \rightarrow 1 \rightarrow 2$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 4$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 3$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$

The implementation is as follow: we still use dfs, because there has no cycles, we have no need to track the visiting state of each node. We generate the possible answer with backtracking technique through the `path` variable to track each state.

```

1 def all_paths(g, s, path, ans):
2     '''generate all paths with backtrack'''
3     ans.append(path[:])
4     for v in g[s]:
5         path.append(v)
6         print(path)
7         all_paths(g, v, path, ans)
8         path.pop()
9         print(path)
  
```

Feed in the above network and run the following code:

```

1 al = [[1], [2], [4], [], [3, 5], [6], []]
2 ans = []
3 path = [0]
4 all_paths(al, 0, path, ans)
  
```

With the printing, we can see the whole process, `path` changes as the description of backtrack.

```

[0, 1]
[0, 1, 2]
[0, 1, 2, 4]
[0, 1, 2, 4, 3]
[0, 1, 2, 4] backtrack
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 5, 6]
[0, 1, 2, 4, 5] backtrack
[0, 1, 2, 4] backtrack
  
```

```
[0 , 1, 2] backtrack
[0 , 1] backtrack
[0] backtrack
```

We can see each state, we can always have a matching backtrack state.

13.2 Solve CSP with Search Pruning

Search Pruning In previous sections, we have learned how backtracking can be applied to enumerating based combinatorial tasks such combination, permutation, and all paths in graph. In this section, we state how backtracking can be optimized with search pruning in CSP. Suppose we are at level 2 with state $s = (s_0, s_1)$, and if we know that this state will never lead to valid solution, we do not need to traverse through this branch but backtrack to previous state at level one. This will end up pruning half of all nodes in the search tree.

Symmetry Exploiting symmetry is another avenue for reducing combinatorial searches, pruning away partial solutions identical to those previously considered requires recognizing underlying symmetries in the search space.

13.2.1 Sudoku

We will start with a sudoku problem due to its popularity in magazines, and we will discuss classical eight queen problem, traveling salesman probelm (TSP), and we leave exercises to such problems such as puzzles, sudoku and so on.

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 13.5: Example sudoku puzzle and its solution

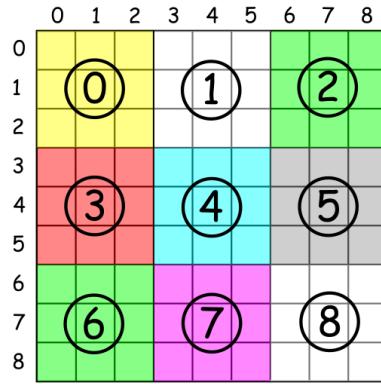


Figure 13.6: Example sudoku

Problem Definition Given a partially filled grid of size $n \times n$, completely fill the grid with number between 1 and n . The constraint is defined as:

1. Each row has all numbers form 1 to ‘n’.
2. Each column has all numbers form 1 to ‘n’.
3. Each sub-grid ($\sqrt{n} \times \sqrt{n}$) has all numbers form 1 to n.

Only all constraint are satisfied can we have a valid candidate. How many possible candidates here? Suppose we have an empty table, the brute force is to try 1 to n at each grid, we have possible solution space of n^{n^2} . How many of them are valid solutions? We can get closer by permutating numbers from 1 to 9 at each row, with $9!^9$ possible search space. This is already a lot better than the first. How to know the exact possible solutions? This site <http://pi.math.cornell.edu/~mec/Summer2009/Mahmood/Count.html> demonstrates that the actual $N = 6670903752021072936960$ which is approximately 6.671×10^{21} possible solutions. This shows that sudo problem is actually NP-hard problem.

Solving Sudoku with Backtracking

Now, let us see how we can use backtrack and search pruning to implement a sudoku solver. Our solution vector s will a length for all empty spots in the given grid. And each item in the vector represents an event for corresponding spot, which might have any number of candidates in the range of 9, just like in the case of permutation, each level’s candidates is constrained by the previous state. In the case of sudoku, we set aside three data structures `row_state`, `col_state`, and `block_state` to validate a candidate. Like any other backtracking algorithm, there are mainly two steps:

1. Initialization: we initialize the three states and prepare data structure to track empty spots.

2. Backtrack: we use DFS to fill in empty spots in a type of ordering.

Step 1: Initialization Assuming we scan the whole grid shown in Fig. 13.5 and find all empty spots that waiting for filling in. We use (i,j) to denote the position of a grid. It correspond position i in `row_state[i]`, and j in `col_state[j]`, and `block_state[i//3][j//3]` for corresponding sub-grid. In this stage, we iterate through the `board` to record these states.

```

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 range
12            (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)

```

Now, with our existing board and state ready to be used in the process of backtracking.

Step 2: Backtracking and Search Pruning In the backtrack, we iterate through the empty spots and for each spot, we iterate through its candidates and fill in one at a time. Before we call recursive function to fill the next one, we record the state. If the sub recursive function returns True

then we just need to return, otherwise, we recover the state and backtrack to previous state.

each time we choose to fill in the spot that with the least number of candidates. To compute the candidates, we can use a set union and set difference $A - (row_state[i] \cup col_state[j] \cup block_state[i//3][j//3])$. We set the time cost for this is $O(9)$, and each time the time cost to pick the best one is $O(9n)$, where n is the number of total empty spots. The total time complexity is $O(n^2)$. Compared with the time complexity of c^n , where c is the average number of candidate for each spot, the time spent here is trivial.

In this problem, backtrack happens if the current path can not lead to valid solution. First, for an empty spot following on the path that has no candidate to choose from (line 5-6), then it is an invalid path, and requires backtrack to line 16. Second, for an empty spot, if none of its candidates can lead to valid solution, as shown in code line 11-16, it backtrack to previous empty spot.

```

1 def solve(self):
2     '''implement solver restricted spot selection and look ahead
3     '''
4     if len(self.unfilled) == 0:
5         return True
6     i, j = min(self.unfilled, key = self._ret_len)
7     option = self.A - (self.row_state[i] | self.col_state[j] |
8                         self.block_state[i//3][j//3])
9     #print(option)
10    if len(option) == 0:
11        return False
12    self.unfilled.remove((i, j))
13    for c in option:
14        self.set_state(i, j, c)
15        if self.solve():
16            return True
17        else:
18            self.reset_state(i, j, c)
19    # no candidate is valid, backtrack
20    self.unfilled.append((i, j))
21    return False

```

Where is the Prunning? If there is only one path that can lead to the final valid answer, this means for other paths, the earlier we find out that it is invalid and backtrack the better. What techniques we have been used here? We could have visit the empty spots in arbitrary ordering instead of each time in our code to choose the spot that has least candidate.

1. **Look Ahead:** The backtrack search works correctly if we do not update the candidates since initialization, and just simply try each candidate and visit the empty spot arbitrarily as we have down in the

naive approach shown in the code. However, a smarter choice is to update candidates for all remaining empty spots for each state change (the remaining empty spots locate at the same row, col, or grid as of the current tempting spot), we are able to make decision global-wisely. If there is a remaining empty spot that end up with no candidate at all. We can detect that the current state is not valid. In the code this is implemented in line 4-7

2. **Most constrained spot selection:** In our solution, we obtain this by comparing the number of possible candidates for all the left empty spots and pick the one has the least possible candidates as shown in line 4. Probabilistically speaking, if there is one spot in the remaining opening spots that has no candidate at all, then we prune this branch instead of wasting our time and effort to first try others spots and then found out that it is an dead end.

```

1 def naive_solve(self):
2     '''implement naivte solver without restricted spot selection
3     or look ahead'''
4     if len(self.unfilled) == 0:
5         return True
6     i, j = self.unfilled.pop()
7     option = self.A - (self.row_state[i] | self.col_state[j] |
8                         self.block_state[i//3][j//3])
9     for c in option:
10        self.set_state(i, j, c)
11        if self.naive_solve():
12            return True
13        else:
14            self.reset_state(i, j, c)
15    # no candidate is valid, backtrack
16    self.unfilled.append((i, j))
17    return False
18 def _ret_len(self, args):
19     i, j = args
20     option = self.A - (self.row_state[i] | self.col_state[j] |
21                         self.block_state[i//3][j//3])
22     return len(option)

```

Time Complexity Assume we have n empty spots, and the number of possible values for each spot are $spot = [a_0, a_1, \dots, a_{n-1}]$. To fill each spot, we need to search the possibility tree. The search tree will have a height of n , at the first level, the width of the tree will be a_0 , second level a_1 , and as such each level will have a total nodes of $\prod_{i=0}^{n-1} a_i = a_i!$. This will result in worst time complexity $O(\sum_{i=0}^{n-1} a_i!)$.

Different Ordering of Empty Slots Let us do an experiment, with the same input of board, we track the time that we use the sorted or unsorted

empty spots and see what is the time difference. The code is provided in colab. The time is 0.025 seconds for unsorted and 0.0005 seconds for sorted.

So, the ordering of the empty slots can make a huge difference to the efficiency. But why and how? Use a simple example, if $spot = [2, 3, 4]$. Compare two ordering, $[2, 3, 4]$ and $[4, 3, 2]$, the total number of nodes for each is $1 + 2 + 2 * 3 + 2 * 3 * 4$ VS $1 + 4 + 4 * 3 + 4 * 3 * 2$. The first ordering has an advantage of $\frac{7}{41}$. Directly, the advantage is not much. Intuitively, to start with the most constrained spot has larger probability ($\frac{1}{a_i}$) to guess it right. Also, as we go deeper of the search tree, and fill in one spot, we add more constraint to the spots later on. Therefore, we always has less branch of each level, larger probability to guess it right, and as it accumulates, we outrun the arbitrary ones in hundreds of times faster.

13.2.2 Eight Queen

Problem Definition Given a chessboard which is of size 8×8 , how many distinct ways to position eight queens on the chessboard such that no two queens threaten each other. According to the chess rules: a queen can move any step either horizontally, or vertically, or diagonally. Which is kind of similar to the rules of sudoku. There is another type of question which asks to return all distinct solutions.

1. Each row can only has one queen.
2. Each column can only has one queen.
3. Each diagonal can only has one queen.

One exemplary solution is shown in Fig. 13.7. The problem can be extend to n -queen problem, which states on any size of $n \times n$ chessboard, how many ways to place n queens that they are mutually non-attacking. Before we rush to the code, we analyze the combinations with constraint. For the n queens, it does not matter about the ordering; like the example, if we switch the positions of them, we will not get another solution. Therefore, it is a combination problem instead of permutation. For combination, we just care the position and for each position, it only differs if there is a queen or not (two choices), while not 9 (no queen plus any of the other queen). We have different ways to arrange these queens:

1. No constraint: (1) if even no constraint of number of queens, for 64 positions, each has two states, this gives us $N = 2^{64}$, (2) put the constraint of only 8 queens, we simply try to come out the possible combination of 8 queens on 8×8 chessboard, it is going to be $N = C_{64}^8 = 4426, 165, 368$.

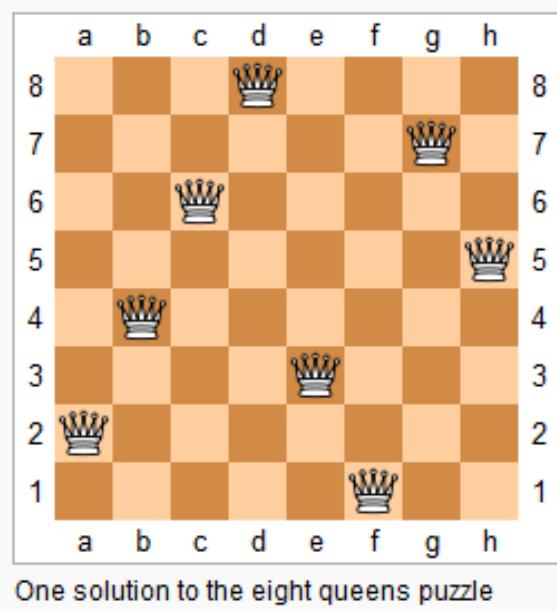
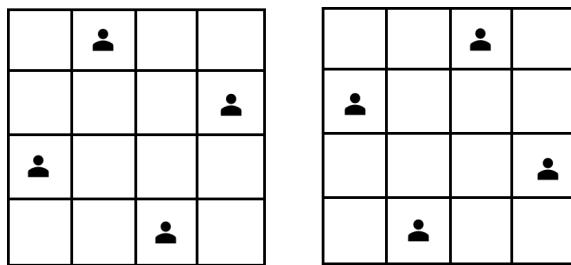


Figure 13.7: An exemplary solution to the eight-queen problem.

2. Add constraint One: Now for the first row, we can have 8 different states, one and only one will have a queen, same for any other rows followed by. We end up with $N = 8^8$.
3. Add constraint Two: If each column can only have one queen, then for the first row, it will have 8 possible states, while the second can only have 7, thus making $N = 8!$, which is less than 10^6 and is possible for programs to run.

Figure 13.8: Solutions shown of 4×4 chessboard

The above analysis reveals that our state vector should be of size of the total number of rows, $S = [None] * 8$, with the index to represent the row in the chessboard, and the value to be an integer from 0 to n-1 to track the column that has a queen. It is similar to a permutation problem. For 4×4 board, we have the following two solutions, represented

with S , it is $S = [1, 3, 0, 2], [2, 0, 3, 1]$. Therefore, our search tree will be of height 8. For edges which represent possible candidate, we can have two ways to generate candidates: (1) easy one that we iterate through all 9 columns for each row and we just need to validate each candidate through `assist_state_tracker`. (2) generate candidate based on previous state vector s .

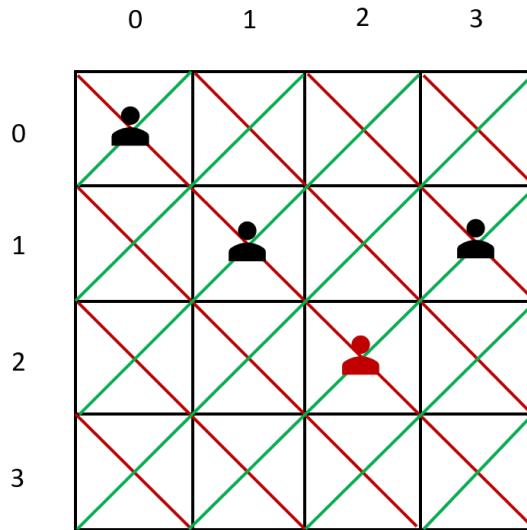


Figure 13.9: Caption

Implementation We use `n_queen` a list whose index indicates the row of the chessboard and the value represents the column that we put the queen in. It starts as an empty list and with a possible maximum length of n . Because in the backtracking, each level represents the row, thus we do not need to track the row state. We need to have `col_state` to track if a column has a queen already. As indicated in Fig. 13.9, for each possible position, we need to check the left and right diagonal if a queen already put there. Here, we use two lists `left_diag` and `right_diag` to track them. For position (r, c) , the position at the `left_diag` will be $(r-1, c-1), (r+1, c+1)$, the rule is bit hidden that $r-c = (r-1)-(c-1) = (r-2)-(c-2)$. For the `right_diag`, it will be $(r-1, c+1), (r+1, c-1)$, thus the same diag has the same value of $(r+c)$. The implementation is as follows:

```

1 def solveNQueens(self, n):
2     """
3         :type n: int
4         :rtype: List[List[str]]
5     """
6     # queen can move: vertically, horizontally, diagonally

```

```

7     col_state = [ False]*n
8     #diag =[False]*n
9     left_diag = [ False]* (2*n-1) # x+y -> index
10    right_diag = [ False]* (2*n-1) # x+(n-1-y) ->index
11    n_queen = [] # to track the positions
12    ans = []
13    board = [[ '.' for i in range(n)] for j in range(n)] #
14      initialize as '.' we can try to flip
15    def collect_solution():
16        board = [[ '.' for i in range(n)] for j in range(n)]
17        for i, j in enumerate(n_queen):
18            board[i][j] = 'Q'
19
20            for i in range(n):
21                board[i] = ''.join(board[i])
22            return board
23
24    def is_valid(r, c):
25        return not (col_state[c] or left_diag[r+c] or right_diag
26 [r+(n-1-c)])
27
28    def set_state(r, c, val):
29        col_state[c] = val
30        #diag[abs(r-c)] = val
31        left_diag[r+c] = val
32        right_diag[r+(n-1-c)] = val
33
34    def backtrack(n_queen, k):
35        if k == n: # a valid result
36            ans.append(collect_solution())
37            return
38        # generate candidates for kth queen
39        for col in range(n):
40            if is_valid(k, col):
41                set_state(k, col, True)
42                n_queen.append(col)
43                backtrack(n_queen, k+1)
44                set_state(k, col, False)
45                n_queen.pop()
46
47    backtrack(n_queen, 0)
48    return ans

```

There is another way to generate candidates based on `n_queen`. At the first row, we have candidates of $[0, 1, 2, 3]$. Assume we choose 1 here, and at row 1, we generate candidates based on previous rows. We remove the ones on the diagonal and columns. The code is implemented as:

```

1 def solveNQueens2(self, n):
2     """
3     :type n: int
4     :rtype: List[List[str]]
5     """
6     n_queen = [] # to track the positions

```

```

7   ans = []
8   board = [[ '.' for i in range(n)] for j in range(n)] #
9     initialize as '.' we can try to flip
10  def collect_solution():
11    board = [[ '.' for i in range(n)] for j in range(n)]
12    for i, j in enumerate(n_queen):
13      board[i][j] = 'Q'
14
15    for i in range(n):
16      board[i] = ''.join(board[i])
17    return board
18
19  def generate_candidate(n_queen, k, n):
20    if k == 0: #the first row, then the candidates row is all
21      columns
22      return set([i for i in range(n)])
23    # generate candidate in kth level based on previous levels
24    candidates = set([i for i in range(n)])
25    for r, c in enumerate(n_queen):
26      if c in candidates:
27        candidates.remove(c)
28      c1 = c-(k-r)
29      if c1 >=0 and c1 in candidates:
30        candidates.remove(c1)
31      c2 = c+(k-r)
32      if c2 < n and c2 in candidates:
33        candidates.remove(c2)
34    return candidates
35
36  def backtrack(n_queen, k):
37    if k == n: # a valid result
38      ans.append(collect_solution())
39    return
40    # generate candidates for kth queen
41    candidates = generate_candidate(n_queen, k, n)
42    for c in candidates:
43      n_queen.append(c)
44      backtrack(n_queen, k+1)
45      n_queen.pop()
46
47  backtrack(n_queen, 0)
48  return ans

```

Symmetry <https://www.aaai.org/Papers/AAAI/2006/AAAI06-257.pdf>.

Start from an easy one, we can observe that we can obtain the second solution of 4×4 chessboard by flipping around the first around the red axis. Assume our first solution is $S = [1, 3, 0, 2]$. We represent the $S = [a_0, a_1, a_2, a_3]$. Now the mirroring relation can be represented as $m_i + a_i = n - 1$, thus $m_i = n - 1 - a_i$. If n is even, then at the first level of backtracking, we can eliminate the second half of candidates, which cut of search cost into half of the previous. If we just need to count the total number of distinct solutions,

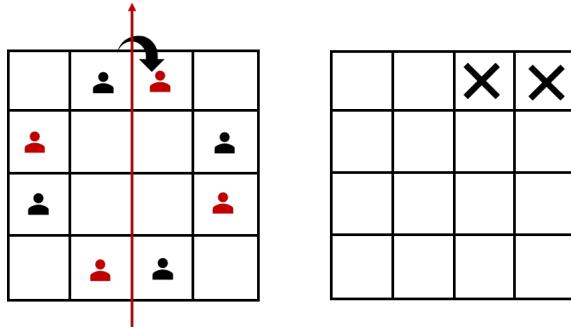


Figure 13.10: Mirroring can cut the search space into half.

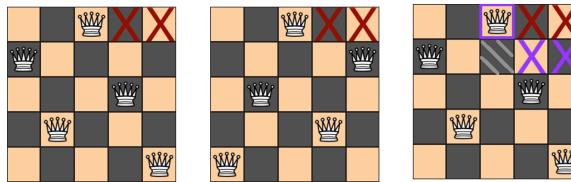


Figure 13.11: Mirroring can cut the search space into half.

we can just double the number we find. The process is shown in Fig. 13.10. For odd one, for the middle spot at the first row, if we follow the same rule as in the even n , some solutions shown in follows will be doubled. So we need to distinguish the middle spot situation in odd n case. If we place a queen in the middle spot of the first row, then for the following $n - 1$ rows, no one will be in the middle column any more. For the second row, we can eliminate the other half of candidates on the right side as shown in Fig. 13.11. Our code become:

```

1 def solveNQueensSymmetry(n):
2     """
3         :type n: int
4         :rtype: List[List[str]]
5     """
6     n_queen = [] # to track the positions
7
8     def generate_candidate(n_queen, s, k, n):
9         if k == s: #apply symmetry
10             candidates = set([i for i in range(n//2)])
11         else:
12             candidates = set([i for i in range(n)])
13
14         for r, c in enumerate(n_queen):
15             if c in candidates:
16                 candidates.remove(c)
17             c1 = c-(k-r)
18             if c1 >=0 and c1 in candidates:
19                 candidates.remove(c1)

```

```

20     c2 = c+(k-r)
21     if c2 < n and c2 in candidates:
22         candidates.remove(c2)
23     return candidates
24
25 def backtrack(n_queen, s, k, ans):
26     '''add s to track the start depth'''
27     if k == n: # a valid result
28         ans += 1
29         return ans
30     # generate candidates for kth queen
31     candidates = generate_candidate(n_queen, s, k, n)
32     for c in candidates:
33         n_queen.append(c)
34         ans = backtrack(n_queen, s, k+1, ans)
35         n_queen.pop()
36     return ans
37
38 # deal with the left half of the first row
39 ans = 0
40
41 ans += backtrack(n_queen, 0, 0, 0)*2
42
43 # deal with the left half of the second row
44 if n%2 == 1:
45     n_queen = [n//2]
46     ans += backtrack(n_queen, 1, 1, 0)*2
47 return ans

```

13.2.3 Travels Salesman Problems (TSP)

13.3 Summary

Backtracking Implementation At first, backtracking might sounds scary and not many books out there did great job to ease it down. If we were to write down a standard template for backtracking, what are the elements? First, imagine that we are building and traverse a tree.

1. **State Vector s :** state vector is something we use to track the solution. In permutation and combination, it is `curr` and `path` in all paths problem. For the sudoku solver, this can be implemented in `unfilled` with the value and position. However, instead, it is easier to directly tracking it on the `board`. The state vector tells us the height of the tree, the candidates for each level is based on the constraint before.
2. **State Map and Candidates:** This is a good example of trading space for efficiency. The constraint of what candidates we have for current node depends on the previous choice, state. We can get previous state by looking at the current result as in permutation in `curr` or

the board in sudoku. However, each lookup took $O(n)$ time to complete. A smarter choice is to set aside another space with boolean or dictionary-like data structure to track the state along with the result data structure. Such as `used` in help of permutation and `row_state`, `col_state`, and `block_state` to assist tracking the constraint in sudoku. Now to lookup if a candidate is possible we only need $O(1)$.

3. Ordering:

4. Look ahead

Backtracking VS Exhaustive Search Backtracking helps in solving an overall problem by incrementally builds candidates (implicit search tree), which is equivalent to finding a solution to the first sub-problem and then recursively attempting to resolve other sub-problems bases on the solution of the first sub-problem. Therefore, **Backtracking can be considered as a Divide-and-conquer method for exhaustive search.** Problems which are typically solved using backtracking technique have such property in common: they can only be solved by trying every possible configuration and each configuration is tried only once(every node one time). A Naive exhaustive search solution is to generate all configurations and “pick” a configuration that follows given problem constraints. Backtracking however works in incremental way and **prunes** branches that can not give a result. It is an optimization over the exhaustive search where all possible(possible still with constraints) configurations are generated and tried. This comparison is called named as **Generating VS Filtering**. Backtracking can be viewed as a smart exhaustive search.

Similarly, the difference between backtracking and DFS is the same. DFS is a searching technique applied on graph or tree data structures (more likely on explicit data structures).

There is an interesting question.



As we explained DFS itself is an incomplete search technique, then why would backtracking search be complete?

13.4 Bonus

Generating combinations, permutation uniformly at random.

13.5 Exercises

13.5.1 Knowledge Check

13.5.2 Coding Practice

Cycle Detection

1. 207. Course Schedule (medium)

Topological Sort

Connected Component

1. 323. Number of Connected Components in an Undirected Graph (medium).
2. 130. Surrounded Regions(medium)
- 3.

Mix

1. 210. Course Schedule II (medium, cycle detection and topological sort).

Backtracking

1. 77. Combinations

```
1 Given two integers n and k, return all possible
2   combinations of k numbers out of 1 ... n.
3 Example:
4
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 ]
```

2. 17. Letter Combinations of a Phone Number

```

1 Given a digit string, return all possible letter
   combinations that the number could represent.
2
3 A mapping of digit to letters (just like on the telephone
   buttons) is given below.
4
5 Input: Digit string "23"
6 Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "
   cf"].
7
8 Note:
9 Although the above answer is in lexicographical order, your
   answer could be in any order you want.

```

3. 797. All Paths From Source to Target (medium).
4. **37. Sudoku Solver (hard).** Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy all of the following rules:

- (a) Each of the digits 1-9 must occur exactly once in each row.
- (b) Each of the digits 1-9 must occur exactly once in each column.
- (c) Each of the the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

Empty cells are indicated by the character “.”

14

Non-Recursive Graph Search

Up till now, we have learned both DFS and BFS in graph search. From the last chapter, we have seen how the backtracking method utilized DFS to solve enumeration and CSP based problems. The type of problem we should solve with backtracking is the one that requires enumeration all possibilities. We have seen that most of the time, backtracking comes with high complexity. If the problem only requires us to find if

14.1 Bidirectional BFS

<http://theoryofprogramming.com/2018/01/21/bidirectional-search/>

14.2

15

Tree-based Search

As the second Chapter related to trees, it is time for us to learn how to complete search a given tree and learn the different applications of trees as mentioned in Chapter ???. Same as a graph data structure, the first thing before thinking about any algorithms is to learn methods to traverse the tree. As in graph, there are two broad ways to iterate all nodes and edges in the tree, Depth-first-search(DFS) and Breath-first-search(BFS). Gladly, due to simpler structure the tree is and it has no cycles, the traverse of a tree is easier to implement than that of a general graph, because there are no cycles in the tree and it is not possible to reach a node from multiple directions.

Tree Traversal We first talk about general traversal of trees, first start with the free trees, which are just simplified version of graph search implementation without the state records. We mainly focus on the implementation of rooted trees and learn both recursive and iterative implementation. These contents will be shown in:

1. Depth-first-search based Tree Traversal in Section 11.3.1.
2. Breath-first-search based Tree Traversal in Section 11.3.2.

Applications We then focus on the applications of trees, mainly with different types of search trees, include binary search tree (BST), segment tree, trie for string and order statistic trees. We would learn how to do search in these different trees and how to implement basic operations to make it a data structure such as insert, delete, and construction.

1. Binary Search Tree in Section ??.

2. Segment Tree in Section ??.
3. Trie for string in Section ??.

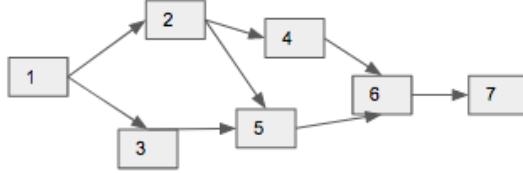


Figure 15.1: Example Graph vs converted tree, where we delete edge $3 \rightarrow 5$ and $5 \rightarrow 6$.

15.1 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.

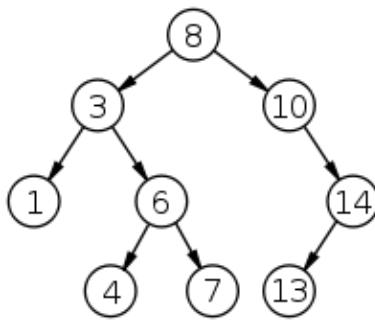


Figure 15.2: Example of Binary search tree of depth 3 and 8 nodes.

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 15.2.

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 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.

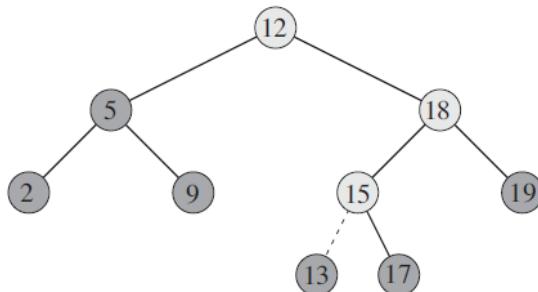


Figure 15.3: The lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

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 15.3. 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]

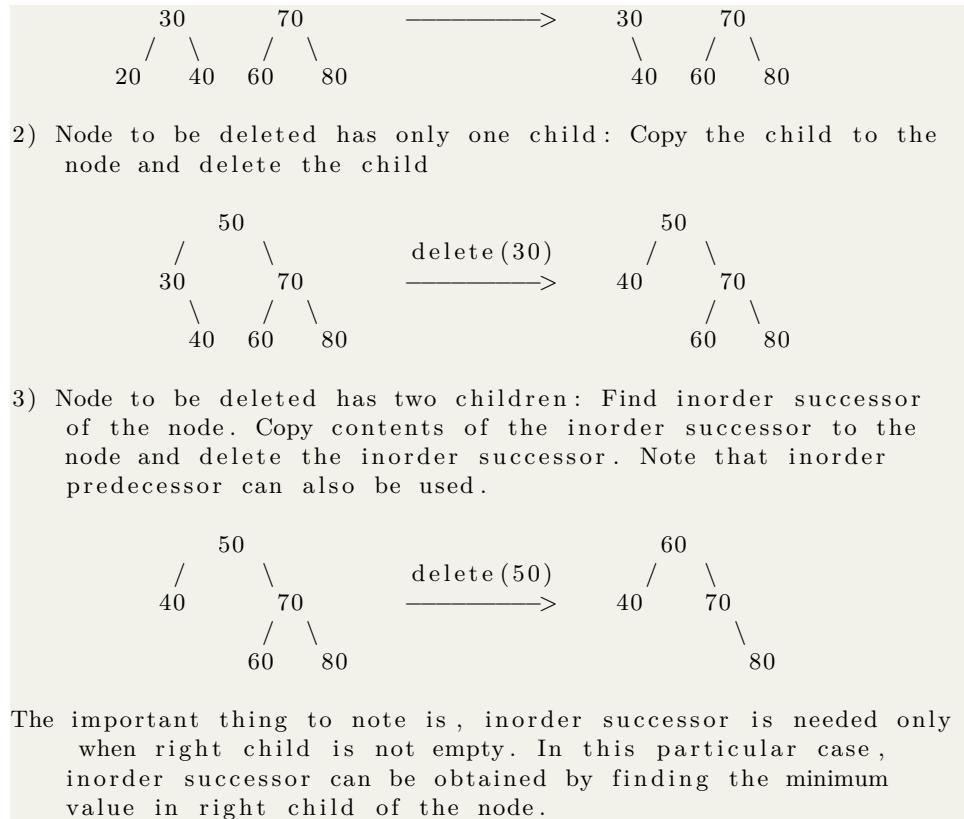
```

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.





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. Inorder Successor is None for the last node in inorder traversal. If our TreeNode data structure has a parent node.

Use parent node: the algorithm 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, successor(12) = 13 = min
   (12.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. 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, successor(12) = 13 = min
   (12.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.
   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.

15.2 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

Table 15.1: 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	$O(lgn)$	$O(n)$

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:

15.1 307. 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 $O(n)$.

```

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     m = (s + e)//2
8     # divide

```

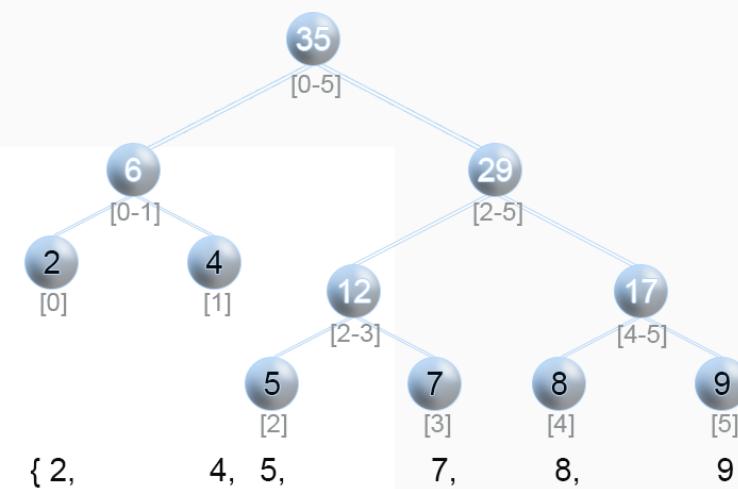
Segmented tree for array $\{2, 4, 5, 7, 8, 9\}$

Figure 15.4: Illustration of Segment Tree.

```

9     left = self._buildSegmentTree(nums, s, m)
10    right = self._buildSegmentTree(nums, m+1, e)
11
12    # conquer
13    node = self.TreeNode(left.val + right.val)
14    node.left = left
15    node.right = right
16    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 15.4, 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 $O(\log n)$.

15.3 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 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 15.5.

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

Compressed Trie

- Compress unary nodes, label edges by strings

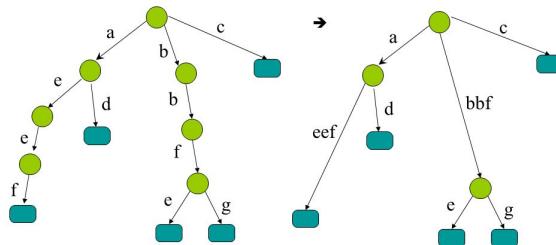


Figure 15.5: Trie VS Compact Trie

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 alphlabetical size, and N is the total number of nodes in the trie structure. The upper bound of N is $n * m$.

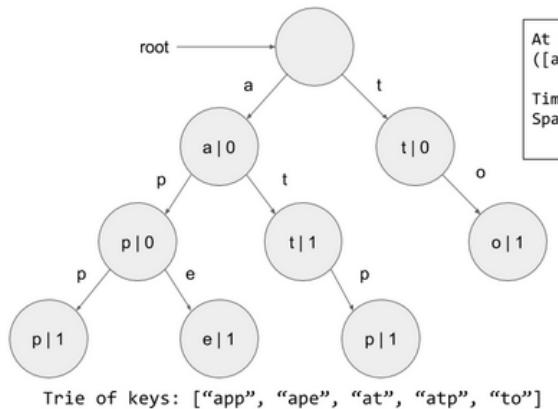


Figure 15.6: Trie Structure

15.1 208. 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 15.5, 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

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

```

15.1 336. 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                    res.append((i, trie.index))
51
52    def palindromePairs(self, words):
53        res = []
54        for i, word in enumerate(words):
55            if not word:
56                continue
57            for j, word2 in enumerate(words):
58                if j == i:
59                    continue
60                if is_palindrome(word + word2):
61                    res.append((i, j))
62        return res

```

```

49     palindrome and the prefix is a word, complete it
50             res.append([i, trie.index])
51     else:
52         # this word is a reverse suffix of other
53         words, combine with those that complete to a palindrome
54         for pali_index in trie.pali_indices:
55             if i != pali_index:
56                 res.append([i, pali_index])
57         if '' in words:
58             j = words.index('')
59             for i, word in enumerate(words):
60                 if i != j and is_palindrome(word):
61                     res.append([i, j])
62                     res.append([j, i])
63
64     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              [(w[::-1], 1, i, len(w))]
7              for i, w in enumerate(words)]) + [len(words) * 2, []]
8
9          #after the sorting, the same string were nearby, one
10         # is 0 and one is 1
11         for i, (word1, rev1, ind1, len1) in enumerate(words):
12             for j in xrange(i + 1, length):
13                 word2, rev2, ind2, _ = words[j]
14                 #print word1, word2
15                 if word2.startswith(word1): # word2 might
16                     be longer
17                     if ind1 != ind2 and rev1 ^ rev2: # one
18                         is reversed one is not
19                         rest = word2[len1:]
20                         if rest == rest[::-1]: result += ([
21                             ind1, ind2],) if rev2 else ([ind2, ind1],) # if rev2 is
22                             reversed, the from ind1 to ind2
23                         else:

```

```

17         break # from the point of view, break
18     is powerful, this way, we only deal with possible
19     reversed,
        return result

```

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.

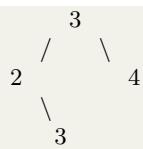
Sorting Lexicographic sorting of a set of keys can be accomplished by building a trie from them, and traversing it in pre-order, printing only the leaves' values. This algorithm is a form of radix sort. This is why it is also called radix tree.

15.4 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:



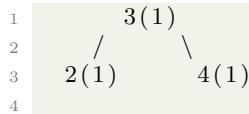
then adding a "3" duplicate key to this tree will result in:



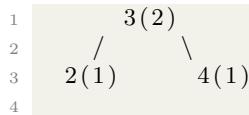
Note that the duplicates are not in contiguous levels.

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.

15.5 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)
```


Part V

Heuristic Search

16

Heuristic Search

A *heuristic* search is a method that:

1. sacrifices completeness to increase efficiency; that it might not always find the best solution, but can be guaranteed to find a good solution in a reasonable time.
2. it is useful in solving tough problems which could not be solved any other way or solutions that take an infinite time or very long time to compute.
3. Classical example of heuristic search methods is the travelling salesman problem.

16.1 Hill Climbing

16.2 Simulated Annealing

16.3 Best-first Search

16.4 The A^* Algorithm

16.5 Graceful decay of admissibility

16.6 Summary and comparison with Complete Search

We will address how to write search algorithms. In particular we will examine:

1. the data structure to keep unexplored nodes. We use a queue (often called a list in many AI books) called OPEN.
2. expansion of a node (or generation of its successors). All the successors of a node can be generated at once (method most commonly used) or they could be generated one at a time either in a systematic way or in a random way. The number of successors is called the branching factor.
3. strategies for selecting which node to expand next. Different algorithms result from different choices (e.g. depth-first when successor nodes are added at the beginning of the queue, breadth-first when successor nodes are added at the end of the queue, etc), test for goal. We will assume the existence of a predicate that applied to a state will return true or false.
4. bookkeeping (keeping track of visited nodes, keeping track of path to goal, etc). Keeping track of visited nodes is usually done by keeping them in a queue (or, better a hash-table) called CLOSED. This prevents getting trapped into loops or repeating work but results in a large space complexity. This is (most often) necessary when optimal solutions are sought, but can be (mostly) avoided in other cases.
5. Keeping track of the path followed is not always necessary (when the problem is to find a goal state and knowing how to get there is not important).

properties of search algorithms and the solutions they find:

1. Termination: the computation is guaranteed to terminate, no matter how large the search space is.
2. Completeness: an algorithm is complete if it terminates with a solution when one exists.
3. Admissibility: an algorithm is admissible if it is guaranteed to return an optimal solution whenever a solution exists.
4. Space complexity and Time complexity: how the size of the memory and the time needed to run the algorithm grows depending on branching factor, depth of solution, number of nodes, etc.

Let's briefly examine the properties of some commonly used uninformed search algorithms.

Depth-First Search

Termination: guaranteed for a finite space if repeated nodes are checked. Guaranteed when a depth bound is used. Not guaranteed otherwise. Completeness: not guaranteed in general. Guaranteed if the search space is

finite (exhaustive search) and repeated nodes are checked. Admissibility: not guaranteed.

Breadth-First Search

Termination: guaranteed for finite space. Guaranteed when a solution exists. Completeness: guaranteed. Admissibility: the algorithm will always find the shortest path (it might not be the optimal path, if arcs have different costs).

Depth-First Search Iterative-Deepening

Termination: guaranteed for finite space. Guaranteed when a solution exists. Completeness: guaranteed. Admissibility: the algorithm will always find the shortest path (it might not be the optimal path, if arcs have different costs).

classes of search algorithms. We can classify search algorithms along multiple dimensions. Here are some of the most common: uninformed (depth-first, breadth-first, uniform cost, depth-limited, iterative deepening) versus informed (greedy, A*, IDA*, SMA*) local (greedy, hill-climbing) versus global (uniform cost, A*, etc) systematic (depth-first, A*, etc) versus stochastic (simulated annealing, genetic algorithms)

Part VI

Advanced Algorithm Design

This part covers two important and advanced paradigm of the four kingdoms: dynamic programming (Chapter 17) and greedy algorithms (Chapter 18).

As we discussed in Divide-and-conquer (Chapter 4), dynamic programming is used for handling situations where in divide and conquer the subproblems overlaps. While, dynamic programming is not a panacea for this problems, it typically applies to certain type of optimization problems which we will explain in details in that chapter. Dynamic programming can be able to decrease the exponential-time complexity into polynomial-time.

Greedy algorithms, just follows Dynamic Programming applies to similar optimization problems with further improvement in efficiency due to making each choice locally optimal.

Dynamic Programming

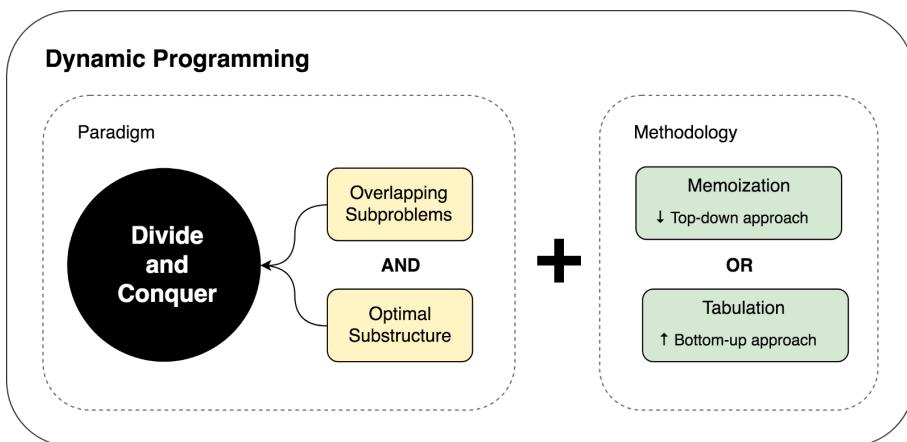


Figure 17.1: Dynamic Programming Chapter Recap

As introduced in Divide-and-Conquer in Chapter 4, dynamic programming is applied on problems that its subproblems show overlapping feature when divided and solved in Divide-and-Conquer manner. We used the recurrence function: $T(n) = T(n - 1) + T(n - 2) + \dots + T(1) + f(n)$, to show its special character. From Chapter 17, we will learn categorized dynamic programming patterns and this formulation will just be one of them.

The naive way which is divide-and-conquer implemented in recursive manner that takes either exponential or polynomial time. The time complexity analysis using recurrence function was also included in Chapter 4.

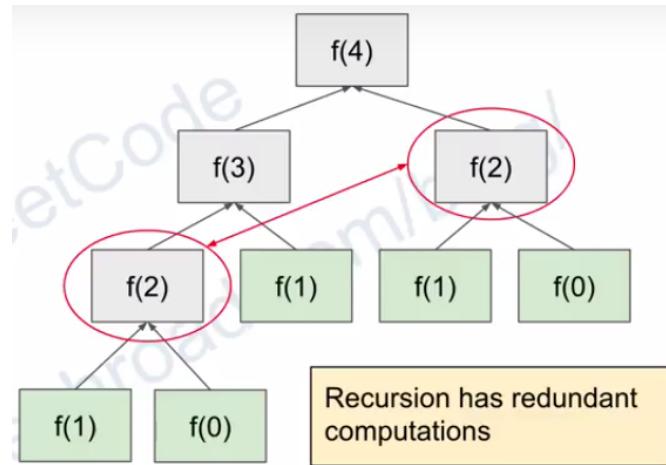
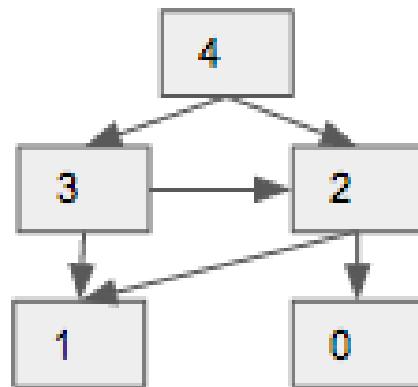


Figure 17.2: Fibonacci number's Recursion Tree

Figure 17.3: Subproblem Graph for Fibononaci Sequence till $n=4$.

Subproblem Graph If we treat each subproblem as a vertex, and the relation between subprblems as edges, we can get a directed subproblem graph. For problems that are divided into non-overlapping subproblems, the subproblem graph would degrade to a tree structure, each subproblem will only have out degree equals to 1. Compared with for the overlapping, some problems would have out degree larger than one, which makes the network a graph instead of a tree structure. Therefore, the above recursive divide-and-conquer solution is actually a Depth-first-search on the subproblem graph/tree. We can also do a Breadth-first-search on the subproblem graph as an alternative choice. To draw the conclusion: *complete search* on the subproblem graph is the most naive way to solve the dynamic programming possible problems.

Complete Search as Naive Solution In Fig. 17.3 shows the corresponding subproblem graph for Fibonacci sequence. We can see for some subproblems, e.g. node 2 and node 1 are both searched twice. Therefore, using the naive complete search method can lead to redundant computation.

Follow the naive complete search, in this Chapter, we will first explain how to evolve the naive complete search solution to the dynamic programming using two examples: fibonacci sequence and longest increasing sequence in Section 17.1. Followed by this, we would have another second characterize the key elements of dynamic programming, and we give examples when dynamic programming used to optimize exponential or polynomial. In the second section, we give generalization: steps to solve the dynamic programming. and more related .

17.1 From Divide-Conquer to Dynamic Programming

Dynamic programming is an optimization methodology that used to improve efficiency from the problem's naive solution—complete search on subprogram graph (Depth-first-search and Breadth-first-search). In this section, we will introduce how we can increase the efficiency of the complete search solution on the subproblem graph. The core here is to solve each subproblem only *once* and saving its solution. Dynamic programming thus uses additional memory to save the computation time; it serves as an example of a *time-space trade-off*. There are two ways in general to do the trade-off, recursive and iterative:

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. In order to avoid the recomputation, we use a hashmap to save the solution to the solved subproblems, and whenever the subproblem is solved, we get its result from the hashmap instead of recompute its solution again. This method follows a top-down fashion.
2. **Bottom-up + Tabulation (iterative):** different from the last solution 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.

The Figure 17.1 record the two different methods, we can use *memoization* and *tabulation* for short. Momoization 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. *Usually,*

dynamic programming solution to a problem refers to the solution with tabulation.

Complexity Analysis To analyze the complexity of the naive DFS based search, we can use method: substitution to solve the recurrence function of this problem as shown in Chapter 4 Section ??.

For the two dynamic programming solutions, the time complexity of the tabulation is more straightforward compared with the recursive memoization. For both of the methods, the core is to number of total subproblems n . For the tabulation, we basically just to analyze how size of the for outmost loop (how many subproblems incurred in the solution) and recurrence relation, how many subproblems we need to reconstruct the solution to current problem, either constant or polynomial order of its problem size. While, in the recursive memoization, no problem is computed twice, therefore, the same rule applies: the number of total subproblem * the number of dependent subproblems.

We enumerate two examples: Fibonacci Sequence (Subsection 17.1.1) and Longest Increasing Subsequence (subsection 17.1.2) in this section to show us in practice how to do the *memoization* and *tabulation*.

17.1.1 Fibonacci Sequence

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 .

Brute Force: Complete Search with DFS. This is simple, because the relation between current state and previous states are directly given. For recursive problem, drawing a *recursive tree structure* can visually assist us to understand better of the structure and the relations. First, it is straightforward to solve the problem in a top-down fashion using recursive call which returns the result of the base cases. And with the returned result of the subtree to build up the result of current node. A nature implementation of the recursive function is DFS traversal. The time complexity can be easily obtained from the tree structure: $O(2^n)$, where the base 2 is the width of the tree.

```

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.

```

DFS+Memoization. From the tree structure, we notice $f(2)$, $f(1)$, $f(0)$ has been computed multiple times, this is due to the overlap between subproblems; for example, for $f(3)$ and $f(4)$, they both need to compute $f(2)$.

To avoid the recomputation, we can use a hashtable to save the solved subproblem. 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)
7     return memo[n]

```

Bottom-up Tabulation. However, the optimized DFS+memoization solution is still facing the same potential threat—stack overflow—as the DFS solution due to the recursive function. If we manage to find the iterative implementation to the DFS+memoization solution, we literally find the solution which is *dynamic programming*. Like the DFS, the result is gained in the returning process which starts from the base case, in the iterative process, we start with base case too. As in Fig 17.3, the topological sort order is [4, 3, 2, 1, 0]. If we visit and construct the solutions for subproblems in order [0, 1, 2, 3, 4] using tabulazied solution to previous smaller subproblems. Specifically, we first need space to store the solved ‘answer’ of each subproblem. Here use a list named dp , it represents the fibonacci number at ‘state’ index (for subproblem 0-index). We then initialize $dp[0]$ and $dp[1]$ because they are the two base cases. Later we use a for loop to reconstruct $f[i]$ from $f[i-1]$ and $f[i-2]$, where $i = [2, n]$. 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]

```

17.1.2 Longest Increasing Subsequence

The Fibonacci sequence we were given the recurrence relation directly. In the example of Longest increasing subsequence, we can learn how to formulate this recurrence relation from the problem.

17.1 300. Longest Increasing Subsequence (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?

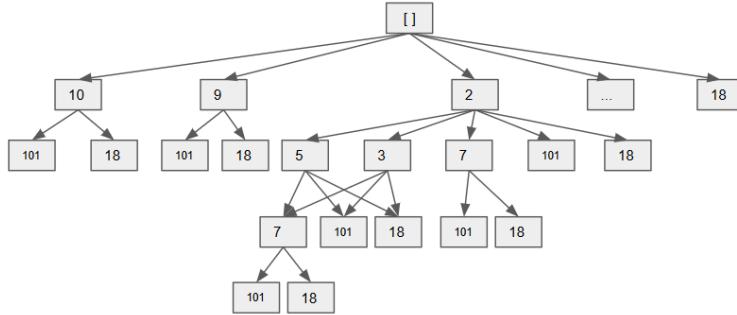


Figure 17.4: State Transfer Tree Structure for LIS, each path represents a possible solution. Each arrow represents a move: find an element in the following elements that's larger than the current node.

Graph Model. If we model each element in the array as a node in the graph, and assume given node μ , if the element v after this one is larger than a node, there will be an edge $\mu \rightarrow v$. The problem is modeled as finding the longest path in the graph, which can be solved in a way doing a DFS or BFS search starts from all possible nodes, and get the longest path.

Recurrence Relation in DFS. For node v and v 's neighbors μ_j , at index i and j respectively, $j > i$, if f is the LIS length. 1) If we use DFS and use Divide-conquer, which returns the length recursively. Then $f(v)$ or $f(i)$ will mean the LIS starts at index i , we could have recurrence relation of $f(i) = \max_j(f(j)) + 1, j > i$. 2) Use BFS, we would not have recurrence relation because it is iterative, and visit its neighbors first. *There is no concept of subproblem, thus there is no recurrence relation for BFS.* Because it is visited level by level, we only get the global answer by visiting all the nodes in the graph. Because of this, there is no recurrence relation in BFS. In BFS, the result needs to be gained from the starting points, we can use an array f to save the maximum LIS length starting from each index. Each time we update the neighbor node's answer with relation function $f(j) = \max(f(j), f(i) + 1)$.

Identify Overlap. Now, if we draw the tree structures of each start-

ing node as shown in Fig 21.1, we can see subproblem which starts at 3, [3, 7, 101, 18] is recomputed multiple times, another one in the subproblem starts at 2, [2, 5, 3, 7, 101, 18]. Therefore if we can memoize the subproblem's solution, we can avoid recomputation in the DFS solution. Unluckily, BFS can not be optimized using memoization even if we figured out the overlap. Now, we give the code of DFS+Memo and DFS.

```
1 # DFS + Memo
2 def recursive(self, nums):
3     """
4         :type nums: List[int]
5         :rtype: int
6     """
7     # helper returns longest increasing subsequence for
8     # nums[idx:]
9     n = len(nums)
10    if not nums: return 0
11    cache = {}
12    def helper(idx):
13        if idx == n-1: return 1
14        if idx in cache: return cache[idx]
15        # Notice at least the answer should be 1
16        ans = 1
17        for i in range(idx+1, n):
18            if nums[idx] < nums[i]:
19                ans = max(ans, 1 + helper(i))
20        cache[idx] = ans
21    return max(helper(i) for i in range(n))
```

```
1 # BFS
2 def lengthOfLIS(self, nums):
3     """
4         :type nums: List[int]
5         :rtype: int
6     """
7     if not nums:
8         return 0
9     lens = [1]*len(nums)
10    q = [i for i in range(len(nums))] # start pos can
11    be any number in nums
12    #lens[0] = 1
13    while q:
14        new_q = set()
15        for idx in q:
16            # search for number that is larger than
17            current
18            for j in range(idx+1, len(nums)):
19                if nums[j] > nums[idx]:
20                    lens[j] = max(lens[j], lens[idx]+1)
21                    new_q.add(j)
22    q = list(new_q)
```

```
21     return max(lens)
```

Dynamic Programming. However, for the above solution, because the use of recursive functions, we might have stack overflow problem. To convert the DFS to iterative, which is the dynamic programming solution. For the DFS and memo, it is equivalent to subproblem as sequence starts at index i. However, in the iterative dynamic programming solution, we can see our subproblems as sequence ends at i. Because in the tabulation method, we use bottom-up approach, therefore the recurrence relation is shown in Eq. 17.1, $\max(LIS(j) + 1)$ where the element j is smaller than current element, or else it only has length one by starting from itself. The whole analysis process is illustrated in Fig 17.5. The function can be written as follows:

Def: LIS(array) := length of LIS ends with array[n-1]		
$LIS([10, 9, 2, 5, 3, 7, 101, 18]) = \max\{$ $LIS([10, 9, 2, 5, 3, 7]) + 1 \quad \leftarrow 4$ $LIS([10, 9, 2, 5, 3]) + 1 \quad \leftarrow 3$ $LIS([10, 9, 2, 5]) + 1 \quad \leftarrow 3$ $LIS([10, 9, 2, 5]) + 1 \quad \leftarrow 2$ $LIS([10, 9]) + 1 \quad \leftarrow 2$ $LIS([10]) + 1 \quad \leftarrow 2$ $\} = 4 \ ([2, 3, 7, 18])$	$LIS([10, 9, 2, 5, 3, 7]) = \max\{$ $LIS([10, 9, 2, 5, 3]) + 1 \quad \leftarrow 3$ $LIS([10, 9, 2, 5]) + 1 \quad \leftarrow 3$ $LIS([10, 9, 2]) + 1 \quad \leftarrow 2$ $\} = 3 \ ([2, 5, 7])$	
$LIS([10, 9, 2, 5, 3, 7, 101]) = \max\{$ $LIS([10, 9, 2, 5, 3, 7]) + 1 \quad \leftarrow 4$ $LIS([10, 9, 2, 5, 3]) + 1 \quad \leftarrow 3$ $LIS([10, 9, 2, 5]) + 1 \quad \leftarrow 3$ $LIS([10, 9, 2]) + 1 \quad \leftarrow 2$ $LIS([10, 9]) + 1 \quad \leftarrow 2$ $LIS([10]) + 1 \quad \leftarrow 2$ $\} = 4 \ ([2, 3, 7, 101])$	$LIS([10, 9, 2, 5]) = \max\{$ $LIS([10, 9, 2]) + 1 \quad \leftarrow 2$ $\} = 2 \ ([2, 5])$	
$LIS([10, 9, 2]) = 1 \ ([2])$		
$LIS([10, 9]) = 1 \ ([9])$		
$LIS([10]) = 1 \ ([10])$		
$ans = \max(LIS(a[0:1]), LIS(a[0:2]), \dots, LIS(a[0:n]))$		

Figure 17.5: The solution to LIS.

$$f(i) = \begin{cases} 1 + \max(f(j)), & 0 < j < i, arr[j] < arr[i]; \\ 1 & \text{otherwise} \end{cases} \quad (17.1)$$

To initialize we set $f[0] = 0$, and the answer is $\max(f)$. 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 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
```

```

10     max_before = 0
11     for j in range(i):
12         if nums[i] > nums[j]:
13             max_before = max(max_before, LIS[j+1])
14             LIS[i+1] = max_before+1
15     return max(LIS)

```

Thus, to get the dynamic programming solution, we need to figure out a way that we will fill out the results for all the states and each state will only dependent on the previous computed states.

17.2 Dynamic Programming Knowledge Base

From the last section, we have learned how to apply *memoization* on DFS based top-down recursive approach and *tabulation* on the bottom-up iterative approach in two problems. However, it is still unclear *when* and *how* we can apply dynamic programming. The main purpose of this section is to inform readers of:

1. the two properties (Section 17.2.1) that an optimization problems must have in order to answer the *when* question: when dynamic programming is to apply?
2. the four key elements(Section 17.2.2) in implementing the dynamic programming solution and to answer the *how* question.
3. Dos and Do nots (Section 17.2.3)
4. Generalization (Section 17.2.4).

17.2.1 Two properties

In order for the dynamic programming to apply, we must characterize two properties: overlapping subproblems and optimal substructure. From Example 17.1, 1) the step of finding the recurrence relation between subproblems in DFS shows the optimal substructure. 2) the step of identifying overlapping shows the overlapping subproblem properties. 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. In dynamic programming, computed solutions to subproblems are stored in table so that they dont have to be recomputed again. So dynamic programming will not be useful there the problems does not show overlapping subproblem property. For example, in the merge sort or binary search, the subproblems are independent from each other and have no overlapping.

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.

17.2.2 Four Elements

From the last example: we can summarize the four key elements to implement the dynamic programming solution to a problem:

1. State: Define what it means of each state, states *the total/the maximum/minimum solution* at position index; This requires us to know how to divide problem into subproblems.
2. State Transfer Function: derive the function that how we can get current state by using result from previous computed state. This requires us to know how to construct the solution of a larger problem from solved smaller problems.
3. Initialization: Initialize the first starting states.
4. Answer: which state or the max/min of all the state is the final result needed.

17.2.3 Dos and Do nots

In this section, we 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 :

1. Optimization: Compute the maximum or minimum value;
2. Counting: Count the total number of solutions;
3. Check if a solution works.

Do nots Dynamic programming can not be used to optimize problems that has $O(n^2)$ or $O(n^3)$ time complexity. However, in the following cases we can not use Dynamic Programming:

1. Be required to obtain or print all solutions, we need to retreat back to the use of DFS+memo(up-down) instead of DP, Draw out the tree structure is necessary and can be extremely helpful;
2. When the input dataset is a set while not an array or string or matrix, 90% chance we will not use DP.

17.2.4 Generalization: Steps to Solve Dynamic Programming

1. Read the question, using Sec 17.2.3 to get a feeling if the dynamic programming applies. Together we analyze the time complexity of the naive solution.
2. Try to draw a tree or graph structure, and see if we can solve the problem using BFS and DFS. This can help us identify the optimal substructure and the overlapping.
3. After we identify that dynamic programming applies, we use the Sec 17.2.2 to implement the algorithm.
4. If we failed the implementation of dynamic programming, retreat to DFS+memoization; if this failed, retreat to DFS or BFS.

Complete Search Assist to Dynamic Programming Also, complete search can be used for us to further validate the solution or even help us to find the recurrence relation between subproblems in some cases.

Top-Down	Bottom-Up
Pro: <ul style="list-style-type: none"> 1. It is a natural transformation from normal Complete Search recursion 2. Compute sub-problems only when necessary (sometimes this is faster) Cons: <ul style="list-style-type: none"> 1. Slower if many sub-problems are revisited due to recursive calls overhead (usually this is not penalized in programming contests) 2. If there are M states, it can use up to $O(M)$ table size which can lead to Memory Limit Exceeded (MLE) for some hard problems 	Pro: <ul style="list-style-type: none"> 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with DP ‘on-the-fly’ technique (see comment in code above) Cons: <ul style="list-style-type: none"> 1. For some programmers who are inclined with recursion, this may be not intuitive 2. If there are M states, bottom-up DP visits and fills the value of <i>all</i> these M states

Table 3.1: DP Decision Table

Figure 17.6: DP Decision

Top-down Vs Bottom-up Dynamic Programming 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 and bottom-up DP in Table 17.6.

17.3 Problems Can be Optimized using DP

Dynamic programming can decrease the complexity that used divide and conquer or searching from exponential level to polynomial, e.g. from $O(2^n)$ or $O(n!)$ to $O(n^m)$, m usually is 2 or 3.

Let us see two more examples: The first one is to optimize a $O(2^n)$ problem, the second one is to optimize a $O(n^3)$ problem.

17.3.1 Example of optimizing Exponential problem

120. Triangle (medium).

```

1 Given a triangle , find the minimum path sum from top to bottom .
   Each step you may move to adjacent numbers on the row below .
2 Example :
3 Given the following triangle :
4
5 [
6 [2] ,
7 [3 ,4] ,
8 [6 ,5 ,7] ,
9 [4 ,1 ,8 ,3]
10 ]

```

The minimum path sum from top to bottom is 11 (i.e., $2 + 3 + 5 + 1 = 11$).

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 from sys import maxsize
2 def minimumTotal(triangle):
3     minSum = maxsize
4     def dfsTraverse(x, y, curSum):
5         nonlocal minSum
6         if x == len(triangle):
7             minSum = min(minSum, curSum)
8             return
9         dfsTraverse(x+1,y, curSum+triangle[x][y])# first
10            adjacent number
11         dfsTraverse(x+1, y+1, curSum+triangle[x][y]) #second
12            adjacent number
13         dfsTraverse(0,0,0)

```

```
12     return minSum
```

In the above solution, the state is a recursive tree, and the DFS traverse all the elements in the tree. To reformulate this problem as dynamic programming, if we use $f[x][y]$ marks the minimum path sum start from (x, y) , then we have this relation $f[x][y] = A[x][y] + \min(f[x+1][y], f[x+1][y+1])$, which gives us a function $T(n) = 2 * T(n - 1)$. We still have $O(2^n)$ time complexity and still encounter LTE error.

```
1 def minimumTotal(triangle):
2     def divideConquer(x, y):
3         if x == len(triangle):
4             return 0
5         return triangle[x][y] + min(divideConquer(x+1, y),
6             divideConquer(x+1, y+1))
6     return divideConquer(0, 0)
```

Recursive and Memoization

Here, for location (x, y) we need to compute $(x + 1, y + 1)$, for location $(x, y + 1)$, $f[x][y + 1] = A[x][y + 1] + \min(f[x + 1][y + 1], f[x + 1][y + 2])$, we compute $(x + 1, y + 1)$ again. So the redundancy exists. However, the advantage of this formate with divide and conquer compared with DFS brute force is that we can use memoization to trade for speed and save complexity. Till now the code is successfully AC.

The time complexity here is propotional to the number of subproblems, which is the size of triangle, $O(n^2)$. This is usually not obvious of its complexity.

```
1 from sys import maxsize
2 def minimumTotal(triangle):
3     memo = [[maxsize for i in range(j+1)] for j in range(len(
4         triangle))]
4     def divideConquerMemo(x, y):
5         #nonlocal memo
6         if x == len(triangle):
7             return 0
8         if memo[x][y] == maxsize:
9             memo[x][y] = triangle[x][y] + min(divideConquerMemo(
10                 x+1, y), divideConquerMemo(x+1, y+1))
11         return memo[x][y]
11     return divideConquerMemo(0, 0)
```

It is normally **Iterative with Space** Now, we do not use the recursive function, the same as the above memoization, we use a memo space f to save the result. This implementation is more difficult compared with the recursive + memoization method. But it is still something managable with practice. The advantages include:

1. It saves the heap space from the implementation of the recursive function.

2. It is easier to get the complexity of the algorithm compared with recursive implementation, simply by looking at its for loops.
3. It is easier to observe the value propagation order, which make it possible to optimize the space complexity.

For the iterative, we have two ways: Bottom-up and top-down. This is compared with your order to fill in the dynamic table. If we use our previous defined relation function $f[x][y] = A[x][y] + \min(f[x+1][y], f[x+1][y+1])$, we need to know the result from the larger index so that we can fill in value at the smaller index. Thus, we need to initialize the result for the largest indices. And we reversely fill in the dynamic table, this is called top-down method, from big index to small. Visually we propagate the information from the end to the front. The final result

On the other side, if we fill in the table from small to larger index, we need to rewrite the relation function to $f[x][y] = A[x][y] + \min(f[x-1][y], f[x-1][y-1])$, this function feedforward the information from the beginning to the end. So we need to initialize the result at $(0,0)$, and the edge of the triangle. following the increasing order, to get value for the larger index, it is bottom-up method.

```

1 #bottom-up
2 from sys import maxsize
3 def minimumTotal(triangle):
4     f = [[0 for i in range(j+1)] for j in range(len(triangle))]
# initialized to 0 for f()
5     n = len(triangle)
#initialize the first point, bottom
6     f[0][0] = triangle[0][0]
#initial the left col and the right col of the triangle
7     for x in range(1, n):
8         f[x][0] = f[x-1][0] + triangle[x][0]
9         f[x][x] = f[x-1][x-1] + triangle[x][x]
10    for x in range(1, n):
11        for y in range(1, x):
12            f[x][y] = triangle[x][y] + min(f[x-1][y], f[x-1][y-1])
13    return min(f[-1])
14
15

```

Top-down with standard space $f[x][y] = A[x][y] + \min(f[x+1][y], f[x+1][y+1])$. Actually for this problem, the top-down method is slightly simpler: we only need to initialize the last row for the state f because for the last row, we cant find its previous state. We directly return result of $f[0][0]$.

```

1 # top-bottom
2 from sys import maxsize
3 def minimumTotal(triangle):
4     f = [[0 for i in range(j+1)] for j in range(len(triangle))]
# initialized to 0 for f()
5     n = len(triangle)
#initial the the last row
6

```

```

7   for y in range(len(triangle[-1])):
8     f[-1][y] = triangle[-1][y]
9   # from small index to large index
10  for x in range(n-2, -1, -1):
11    for y in range(x, -1, -1):
12      f[x][y] = triangle[x][y] + min(f[x+1][y], f[x+1][y
13      +1]) #get result for larger state from smaller state
14  return f[0][0]

```

Or we have top down and save the space by reusing triangle matrix. The code is even simpler and we can have $O(1)$ space complexity.

```

1 # top-down and no extra space
2 def minimumTotal(triangle):
3   n = len(triangle)
4   # from small index to large index
5   for x in range(n-2, -1, -1):
6     for y in range(x, -1, -1):
7       triangle[x][y] = triangle[x][y] + min(triangle[x+1][
8         y], triangle[x+1][y+1]) #get result for larger state from
9       smaller state
10  return triangle[0][0]

```

17.3.2 Example of optimizing Polynomial problem

53. Maximum Subarray (Easy). 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.

Brute Force solution: The brute force solution of this problem is to use two for loops, one pointer at the start position of the subarray, the other point at the end position of the subarray. Then we get the maximum sum of these subarrays. The time complexity is $O(n^3)$, where we spent $O(n)$ to the sum of each subarray. However, if we can get the sum of each subarray with $O(1)$. Then we can lower the complexity to $O(n^2)$. Here one solution is to get $sum(i+1) = sum(i) + nums[i+1]$.

```

1 from sys import maxsize
2 def maximumSubarray(nums):
3   if not nums:
4     return 0
5   maxValue = -maxsize
6   for i, v in enumerate(nums):
7     accSum = 0
8     for j in range(i, len(nums)):
9       #accSum = sum(nums[i:j+1])
10      accSum += nums[j]
11      maxValue = max(maxValue, accSum)
12  return maxValue

```

Another way that we can get the sum between i, j in $O(1)$ time with formula $sum(i, j) = sum(0, j) - sum(0, i)$ by using $O(n)$ space to save the sum from 0 to current index.

solution: Divide and Conquer To further improve the efficiency, we use divide and conquer, where we divide one array into two halves: the maximum subarray might located on the left size, or the right side, or some in the left side and some in the right size, which crossed the bound. $T(n) = max(T(left), T(right), T(cross))$, 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 $O(nlgn)$. With this solution, we use $O(lgn)$ space for the recursive function stack space.

```

1 def maxSubArray(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: int
5     """
6     def getCrossMax(low, mid, high):
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        #right_low, right_high, right_sum = maxSubarray(mid+1, high)
30        #[mid+1, high]
31        rslt.append(maxSubarray(low, mid)) #[low, mid]
32        #cross_low, cross_high, cross_sum = getCrossMax(low,
33        mid, high)
34        rslt.append(getCrossMax(low, mid, high))
35        return max(rslt, key=lambda x: x[2])
36    return maxSubarray(0, len(nums)-1)[2]
```

Dynamic Programming: Using dynamic programming: the f memorize

the maximum subarray value till j , the state till i we can get the result from previous state $i - 1$, the value of current state depends on the larger one between $f[i - 1]$ plus the current element and the current element, which is represented as $f[i] = \max(f[i - 1] + \text{nums}[i], \text{nums}[i])$. This would give us $O(n)$ time complexity and $O(n + 1)$ space complexity. The initialization is $f[0] = 0$, and the answer is $\max(f)$.

```

1 from sys import maxsize
2 def maximumSubarray(nums):
3     if not nums:
4         return 0
5     f = [-maxsize]*(len(nums)+1)
6     for i, v in enumerate(nums):
7         f[i+1] = max(f[i]+nums[i], nums[i]) #use f[i+1] because
8         we have n+1 space
9     return max(f)

```

However, here since we only need to track $f[i]$ and $f[i + 1]$, and keep current maximum value, so that we do not need to use any space.

```

1 from sys import maxsize
2 def maximumSubarray(nums):
3     if not nums:
4         return 0
5     f = 0
6     maxValue = -maxsize
7     for i, v in enumerate(nums):
8         f = max(f+nums[i], nums[i]) #use f[i+1] because we have
9         n+1 space
10        maxValue = max(maxValue, f)
11    return maxValue

```

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

```

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.

```

1 def partition(self, s):
2     """
3         :type s: str

```

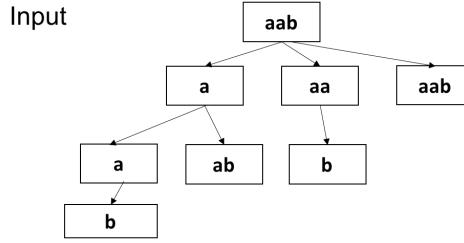


Figure 17.7: State Transfer for the panlindrom splitting

```

4     :rtype: List[List[str]]
5     """
6     #s=="bb"
7     #the whole purpose is to find pal, which means it is a DFS
8     def bPal(s):
9         return s==s[::-1]
10    def helper(s, path, res):
11        if not s:
12            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
13    res = []
14    def helper(start, path, res):
15        if start==len(s):
16            res.append(path)
17        for i in range(start, len(s)):
18            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.

17.3.3 Single-Choice and Multiple-Choice State

Generally speaking, given a sequence, if the final solution can be constructed from a subproblem that choose a certain element from its current state, then it is a single state dynamic programming problem. For example, the Longest increasing subsequence problem. On the other hand, for the subproblem, we can have multiple choice, and the optimal solution can be dependent on all of these different states. For example, 801. Minimum Swaps To Make Sequences Increasing. Knowing how to solve problems that has multiple-choice state is necessary.

Two-Choice State

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] < ... < 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 will
be in the range [1, 1000].
20 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], i
11 +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        return count
20
21    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 muliple 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 the
6         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+1,
13 memo, False), dfs(B[i], A[i], i+1, memo, True)+1)

```

```

12         return memo[(swapped, i)]
13     count = sys.maxsize
14
15     if A[i]>a and B[i]>b: #not swap
16         count = min(count, dfs(A[i], B[i], i+1, memo,
17                               False))
17     if A[i]>b and B[i]>a: #swap
18         count = min(count, dfs(B[i], A[i], i+1, memo,
19                               True) +1)
20     memo[(swapped, i)] = count
21
22     return memo[(swapped, i)]
23
24     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 and i
11            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 swap
14            dp_swap[i] = min(dp_swap[i], dp_swap[i-1]+1)
15        if A[i] > B[i-1] and B[i] > A[i-1]: # i-1 not swap, i
16            swap
17            dp_swap[i] = min(dp_swap[i], dp_not[i-1]+1)
18            # if i-1 swap, it means the first case, current need
19            to not to swap
20            dp_not[i] = min(dp_not[i], dp_swap[i-1])
21
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.

17.4 Time Complexity Analysis

Substitution Method

The substitution method comprises two steps to solving the recurrence and the either the upper bound or lower bound time complexity.

1. Guess the form of the solution.

2. Use mathematical induction to find the constants and show that the solution works.

For example, given a recurrence equation as follows:

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \quad (17.2)$$

We guess the upper bound time complexity is $O(nlg n)$, which is to say we need to prove that $T(n) \leq cnlg n$ for an appropriate choice of the constant $c > 0$. We start by assuming that this holds true for some smaller $m < n$, here we let $m = \lfloor n/2 \rfloor$, which yields $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor lg(\lfloor n/2 \rfloor)$, then we substitute $T(\lfloor n/2 \rfloor)$ into Eq. 17.2, we have the following inequation:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor lg(\lfloor n/2 \rfloor)) + n \\ &\leq cnlg(n/2) + n \\ &\leq cnlg n - cnlg 2 + n \\ &\leq cnlg n - cn + n \\ &\leq cnlg n, \text{ if } c \geq 1, \end{aligned} \quad (17.3)$$

Unfortunately, it could be difficult to come up with an asymptotic guess (that as close as the above one). It takes experience. Fortunately, we still have two ways that help us to make a better guess.

1. We can draw a recursive tree that visually helps us make a heuristic guess.
2. We can make a guess to prove the loose upper and lower bounds on the recurrence and then reduce the range of uncertainty.

Also, for our guess, if we directly substitute we can not always prove the induction. Consider the recurrence:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1, \quad (17.4)$$

A reasonable and logical guess for this recurrence is $T(n) = O(n)$, so that we obtain

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1, \\ &= cn + 1 \end{aligned} \quad (17.5)$$

However, the induction is not exactly having cn as an upper bound, but it is constant close enough. If we are just in the coding interview, we do not need to bother too much about slightly subtleties.

17.5 Exercises

17.5.1 Knowledge Check

17.5.2 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 1. Right -> Right -> Down -> Down
2 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            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, costs[0][0] is the cost of painting house 0 with color red; 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 minCost[i] = till i the mincost for each color for color 0: paint 0 [0] = min(minCost[i-1][1], minCost[i-1][2]) + costs[i][0]

paint 1 [1]

minCost[i] = [0,1,2], i for i in [0,1,2]

answer = min(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]
22                                  for pre_cor in pres]) + costs[i-1][c]
23    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)]
8     minCost[0] = [0]*k
9     minCost[1] = [cost for cost in costs[0]]
10    colorSet = set([i for i in range(k)])
11    for i in range(2, len(costs)+1):

```

```

12     for c in range(k):
13         #previous color
14         pres = list(colorSet-set([c]))
15         minCost[i][c] = min([minCost[i-1][pre_cor]
16             for pre_cor in pres])+costs[i-1][c]
17         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
13     +1)]
14     same = k
15     diff = k*(k-1)
16     for i in range(3,n+1):
17         pre_diff = diff
18         diff = (same+diff)*(k-1)
19         same = pre_diff
20     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 \wedge 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 = "aadbcbcbcac"
2 Output: true

```

Example 2:

```

1 Input: s1 = "aabcc" , s2 = "dbbca" , s3 = "aadbcccac"
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    //search for right_min
32    localMin = nums[size - 1];
33    for (int i = size - 2; i >= 0; i--) {
34        localMin = Math.min(nums[i], localMin + nums[i]);
35        right_min[i] = Math.min(right_min[i + 1], localMin);
36    }
37    //search for separate position
38    int diff = 0;
39    for (int i = 0; i < size - 1; i++) {
40        diff = Math.max(Math.abs(left_max[i] - right_min[i + 1]), diff);
41        diff = Math.max(Math.abs(left_min[i] - right_max[i + 1]), diff);
42    }
43    return diff;
44}

```

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 ith element. The function is as follows.

$$\min_local[i] = \begin{cases} \min(\min_local[i-1] * \text{nums}[i], \text{nums}[i]), & \text{nums}[i] < 0; \\ \min(\max_local[i-1] * \text{nums}[i], \text{nums}[i]) & \text{otherwise} \end{cases} \quad (17.6)$$

$$\max_local[i] = \begin{cases} \max(\max_local[i-1] * \text{nums}[i], \text{nums}[i]), & \text{nums}[i] > 0; \\ \max(\min_local[i-1] * \text{nums}[i], \text{nums}[i]) & \text{otherwise} \end{cases} \quad (17.7)$$

```
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 (
4               price = 5), profit = 5-1 = 4.
5               Then buy on day 4 (price = 3) and sell on day
6               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 $f[i-A[i]]+V[i]$.

17.6 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 17.8: Summary of different type of dynamic programming problems

18

Greedy Algorithms

From the catalog: Greedy Algorithm listed on LeetCode, 90% of the problems are medium and hard difficulty. Luckily, we can see the frequency is extremely low for each single problem available on LeetCode. Even the most popular coding instruction book *Cracking the Coding Interview* does not even mention Greedy Algorithm. So, we do not need to worry about failing the interview due to this type of questions, because it is the most unlikely type of questions in the interview. Probably your interviewer would not like to try it out himself or herself.

But still, to complete the blueprint of this book, we are still going to learn the concepts of greedy algorithms, and try to solve the problems on the LeetCode as practice.

18.1 From Dynamic Programming to Greedy Algorithm

It is important to know that Greedy Algorithm just as dynamic programming still follow the methodology of Divide and Conquer. In greedy algorithm, we divide the ultimate problem into subproblems, we start from solving the smallest subproblem in our practice and use the solution from the subproblem to reconstruct the solution to the upper-level subproblem till we solve our ultimate problem.

The difference of Greedy Algorithm compared with the Dynamic programming is that with dynamic-programming, at each step we make a choice which usually dependent on and by comparing between the multiple solutions of the recurrence function; while in for the greedy algorithm, we just need to consider one choice-the greedy choice-and by solving this chosen

subproblem and using its corresponding result we can recursively construct the optional solution. In dynamic programming, we solve subproblems before making the first choice and usually processing in a *bottom-up fashion*, a greedy algorithm makes its first choice before solving any subproblems, which is usually in *top-down fashion*, reducing each given problem instance to a smaller one.

It is not easy to develop an instinct to know if greedy solution can solve a specific problem. We can always start from solving a problem which can be solved with dynamic programming. Recall the previous chapter that for dynamic programming we need to construct the recurrence equation. After we figured out how to solve a problem with dynamic programming, and we can brainstorm further to check if this problem can be further optimized with going to greedy algorithm.

Due to the special relationship between greedy algorithm and the dynamic programming: "beneath every greedy algorithm, there is almost always a more cumbersome dynamic programming solution", we can try the following six steps to solve a problem which can be potentially solved by making greedy choice:

1. *Divide* the problem into subproblems, including one problem and the remaining subproblem.
2. Determine the optimal *substructure* of the problems. and formulating a recurrence function.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Validate the rightness of the greedy choice.
5. Write either a recursive or an iterative implementation.

18.2 Hacking Greedy Algorithm

More generally, after we get more experienced with greedy algorithm, we can skip the dynamic programming part and use the following steps:

1. Cast the optimization problem as one in which we make a choice and are left with only one subproblem to solve.
- 2.

Two key ingredients that can help us identify if a greedy algorithm will solve a specific problem: Greedy-choice property and optimal substructure.

18.2.1 Greedy-choice Property**18.2.2 Prove the Correctness of Greedy Algorithms**

<http://www.cs.cornell.edu/courses/cs482/2007su/exchange.pdf>.

Greedy

[Subscribe](#) to see which companies asked this question

You have solved **7 / 38** problems.

[Show problem tags](#)

#	Title	Acc
✓ 122	Best Time to Buy and Sell Stock II	49.4
860	Lemonade Change	49.3
455	Assign Cookies	47.5
874	Walking Robot Simulation	28.4
861	Score After Flipping Matrix	67.5
✓ 763	Partition Labels	65.5
✓ 406	Queue Reconstruction by Height	57.2
484	Find Permutation 🔒	56.2
651	4 Keys Keyboard 🔒	49.6
714	Best Time to Buy and Sell Stock with Transaction Fee	47.8
392	Is Subsequence	45.1
452	Minimum Number of Arrows to Burst Balloons	44.7
621	Task Scheduler	42.6
738	Monotone Increasing Digits	41.2
435	Non-overlapping Intervals	41.0
✓ 253	Meeting Rooms II 🔒	40.3
881	Boats to Save People	39.8
870	Advantage Shuffle	39.8
✓ 767	Reorganize String	39.2

Figure 18.1: Screenshot of Greedy Catalog, showing the frequency and difficulty of this type in the real coding interview

Part VII

Advanced and Special Topics

In this part, we would include more detailed topics such as advanced graph algorithms, string process, dynamic programming, backtracking.

19

Advanced Graph Searching Algorithms

This chapter is built upon on Part IV, specifically on Chapter 11, namely graph searching algorithms that might combine the basic searching strategies and advanced algorithm Design methodologies such as dynamic programming and greedy algorithms in order to solve the problems revealed in this Chapter.

Application of Basic Graph Search:

1. Cycle Detection (Section 19.1),
2. Topological Sort(Section 19.2),
3. Connected Components(Section 19.3).

The first several sections, we talk about the advanced applications of DFS, including Connected Components in Graph (Section 19.3), Topological Sort(), Minimum Spanning Tree (MST) ??.

As we will see, Prim's MST algorithms, Dijkstra algorithm to find shortest path are a combination of BFS and greedy algorithms. The Floyd-Warshall algorithm, which finds shortest paths between all pairs of vertices is a dynamic programming algorithm.

19.1 Cycle Detection

Given a path v_0, v_1, \dots, v_k in graph G :

1. If G is directed, we say it forms a **cycle** if $v_0 = v_k$ and the path contains at least one edge. The cycle is **simple** if in addition v_1, v_2, \dots, v_k are distinct. A self-loop is a cycle of length 1.
2. If G is undirected, this path forms a cycle only if $v_0 = v_k$ and $k \geq 3$.

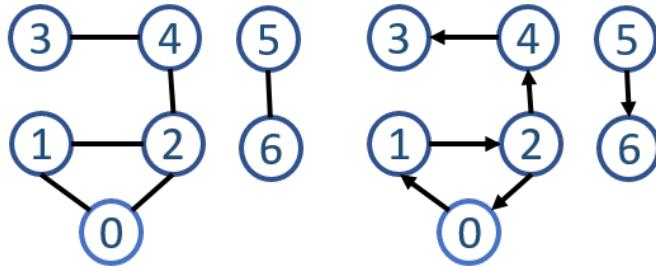


Figure 19.1: Example of Cycle Detect in both Undirect and Directed Graph

Back Edge and Cycle Detection Recall the back edge we categorized in the DFS graph search, which is an edge (u, v) connecting a vertex u back to an ancestor v in the depth-first tree. When a back edge appears in the depth-first tree, if the graph is directed, we then find one cycle. In the undirected graph, we need to do a little bit extra check because $k \geq 3$. In the undirected graph, we need to distinguish between cycle of two nodes and cycle of more nodes. If this path includes two nodes as of v_0, v_1, v_2 , when we are visiting v_1 , the predecessor of v_1 , which is v_0 will be the same as of v_2 which is a node with gray state. So, in addition to a normal DFS based cycle detection for the directed graph, we track the predecessor of the current visiting vertex.

Implementation for Directed Graph And we have a function `hasCycle` which is pretty much the same as the DFS we have implemented before other than it has the return value and extra condition check on the back edge.

```

1 def hasCycle(g, s, state):
2     '''convert dfs to check cycle'''
3     state[s] = STATE.gray # first be visited
4     for v in g[s]:
5         if state[v] == STATE.white:
6             if hasCycle(g, v, state):
7                 return True
8         elif state[v] == STATE.gray: # a back edge
9             return True
10        else:
11            pass
12        state[s] = STATE.black # mark it as complete
13
14    return False

```

Then, We iterate all vertices and do a DFS from a vertex if it is unvisited yet in the `cycleDetect` main function.

```

1 def cycleDetect(g):
2     '''cycle detect in directed graph'''
3     n = len(g)

```

```

4 state = [STATE.white] * n
5 for i in range(n):
6     if state[i] == STATE.white:
7         if hasCycle(g, i, state):
8             print('cycle starts at vertex ', i)
9             return True
10    return False

```

We run an example with our exemplary directed graph, we have the following output:

```
cycle starts at vertex 0
```

Implementation for Undirected Graph First, the `hasCycle` we add another variable `p` to track the predecessor. `p` will first be initialized to -1, because no node will be named as of the same.

```

1 def hasCycle(g, s, p, state):
2     '''convert dfs to check cycle'''
3     state[s] = STATE.gray # first be visited
4     for v in g[s]:
5         if state[v] == STATE.white:
6             if hasCycle(g, v, s, state):
7                 return True
8         elif state[v] == STATE.gray and v != p: # aback edge
9             return True
10        else:
11            pass
12    state[s] = STATE.black # mark it as complete
13
14    return False

```

Then the same Cycle Detect:

```

1 def cycleDetect(g):
2     '''cycle detect in directed graph'''
3     n = len(g)
4     state = [STATE.white] * n
5     for i in range(n):
6         if state[i] == STATE.white:
7             if hasCycle(g, i, -1, state):
8                 print('cycle starts at vertex ', i)
9                 return True
10    return False

```

 How to find all cycles?

19.2 Topological Sort in Directed Acyclic Graph

Topological Sort or topological ordering of a **Directed Acyclic Graph (DAG)** is a linear ordering of the vertices u comes before v if edge $(u \rightarrow v)$ exists in the DAG. Every DAG has at least one or more topological sorts. We do not use topological sort either undirected graph or directed cyclic graph because for cycles, it is impossible for us to decide who the ordering of vertices within the cycle. For example, the topological sort of the Fig 19.2

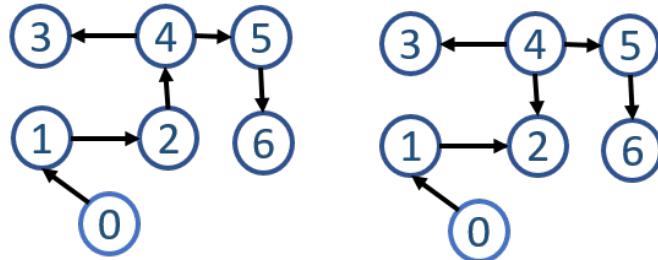


Figure 19.2: Example of Topological Sort

can be $[0, 1, 2, 4, 3, 5, 6]$ and $[0, 1, 2, 4, 5, 6, 3]$. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges). Because for the topological sort must cover all vertices, so that we need to run DFS on all vertices.

Discovery ordering VS finishing ordering As we have discussed in the DFS section, there are two orderings of the nodes, either according to the discovery time or the finishing time. The above topological sort orderings are the orderings of discovery and the reversed finishing ordering each. For this example, we might have a false conclusion that both ordering works. However this is not the case. Look at Fig. 19.2, we revered the edge $(2 \rightarrow 4)$ to $(4 \rightarrow 2)$. With this ordering, the discovery order is $[0, 1, 2, 3, 4, 5, 6]$, which is wrong because 4 need to come front of 3.

Implementation The implementation is trivial, we just need BFS run potentially on all vertices by using a for loop to run each unvisited node as a source node. The only thing is to reverse the `complete_order`.

```

1 def topo_sort(g):
2     n = len(g)
3     orders, complete_orders = [], []
4     colors = [STATE.white] * v
5     for i in range(n): # run dfs on all the node
6         if colors[i] == STATE.white:
7             dfs(g, i, colors, orders, complete_orders)
8

```

```
9 return orders, complete_orders[::-1]
```

Call `topo_sort` on the first and second graph, we will have the sorted ordering as:

```
1 [0, 1, 2, 4, 5, 6, 3]
2 [4, 5, 6, 3, 0, 1, 2]
```

Why it Works? Why the reversed finishing order is one of the topological sort? In DAG, given any edge $(u \rightarrow v)$, because the graph is acyclic which means the DFS will yield no back edge. This also means that when u is explored, v will not be gray because that would cause a cycle. Now, there are two cases: (1) v can be white, which makes the v a descendant of u , in the DFS, a descendant always finishing earlier than the predecessor. (2) v can be black. This means v is already been fully visited, then its finishing time is earlier than u . Therefore, in DAG, for each edge $(u \rightarrow v)$, the DFS finishing order can always put v ahead of u . If we reverse this ordering, we get on of the topological sort.

Application Topological sort is used to schedule events that having orders.

19.3 Connected Components

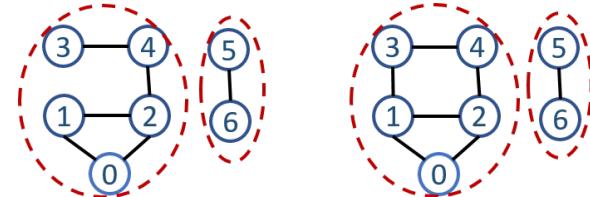


Figure 19.3: The connected components in undirected graph, each dashed red circle marks a connected component.

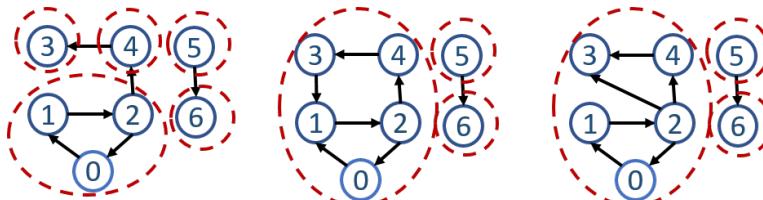


Figure 19.4: The strongly connected components in directed graph, each dashed red circle marks a strongly connected component.

In graph theory, a connected component (or just component) is used to refer a subgraph of an undirected graph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph. For example, the undirected graph in Fig. 19.3 has two connected components: $\{0, 1, 2, 3, 4\}$ and $\{5, 6\}$. In the directed graph, the term **Strongly Connected Components (SCC)** or **diconnected** are used to refer to the same relation, where in a strongly connected component any vertices are reachable to each other by paths, that is there is a path in each direction between each pair of vertices of the graph. For example, the directed graph in Fig. 19.4, there are in total three strong connected components: $\{0, 1, 2\}$, $\{3\}$, $\{4\}$, $\{5\}$, and $\{6\}$. Unlike in the undirected graph, vertex 5 and 6 can only reach to itself and 5 is not reachable from 6. The connection of connected or strongly connected components forms a partition of the set of vertices of G .

There is another notion of connected components for directed graphs, called *weakly connected* if it is connected after ignoring the direction of the edges. Therefore, in Fig. 19.4, for the weakly connected components, they are the same as of the connected components in the previous undirected graphs. To detect them takes a trivial step ahead of the detection of connected components in the undirected graph, converting the DAG to undirected graphs.

Relation to Cycles Observing the above examples, in the undirected graph, the cycle we detected in Section 19.1 is $\{0, 1, 2\}$, and it is obvious to distinguish the connected components with cycles. For cycles, they must be connected components, while not the other way around. Look at the directed graph example, the relation of cycles and SCCs is more indistinguishable. The cycle $\{0, 1, 2, 0\}$ is a SCC in the first graph, and in the second graph, the scc 1 has two cycles: has $\{0, 1, 2, 0\}$ and $\{1, 2, 3, 4, 1\}$, and they share two common vertices 1, and 2.

19.3.1 Finding Connected Components

Search Tree Finding connected components for an undirected graph is trivial. We can utilize either the breath-first tree and the depth-first tree. In the undirected graph, these two types of trees denotes that all nodes they have are reachable between each other. Therefore, a DFS or BFS begins at some vertex u will find the entire connected components containing u . To find all the connected components, we simply need to loop through all of its vertices, start a new DFS or BFS whenever the loop reaches a vertex that has not already been included in a previously found connected components. DFS can be used to mark the connected components and count the total number of connected components. The time complexity will be $O(|V| + |E|)$ and the space complexity will be $O(|V|)$. We give the code as follows:

```

1 def bfs(g, s, state):
2     state[s] = True
3
4     q, orders = [s], [s]
5     while q:
6         u = q.pop(0)
7
8         for v in g[u]:
9             if not state[v]:
10                 state[v] = True
11                 q.append(v)
12                 orders.append(v)
13
14     return orders
15
16 def connectedComponent(g):
17     n = len(g)
18     ccs = []
19     state = [False] * n
20     for i in range(n):
21         if not state[i]:
22             ccs.append(bfs(g, i, state))
23
24 return ccs

```

Finding strongly connected components is a bit more tricky. We will derive and reason one algorithm called **Kosaraju's SCCs**.

Implementation with Disjoint Sets The problem of computing the connected components using graph search arises when new edges are added to the graph afterwards. Implementation using disjoint set data structure can fit better for dynamic graph, and the previous graph search solution is better for statistic graph.

19.3.2 Finding Strongly Connected Components

We can add more properties of SCCS. We have observed part of the relation of between cycles and SCCs. In graph theory, if each SCC is contracted to a single vertex, the resulting graph is a directed acycle graph, the **condensation** of G. For example, mark each SCC with red dashed circles in Fig. 19.6. Here, we define the contracted DAG as **component graph** and denote it as $G^{SCC} = (V^{SCC}, E^{SCC})$. The vertex set $V^{SCC} = \{v_1, v_2, \dots, v_k\}$, each v_i represented a strongly connnected component C_i . If G contains a directed edge $(u \rightarrow v)$ for $u \in C_i$ and $v \in C_j$, then there is an edge $(v_i \rightarrow v_j) \in E^{SCC}$. With the definition, we redraw our example to component graph.

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, we can SCCs algorithms can indirectly detect cycles. If there exists nontrivial SCC, there exists of cycle in the directed graph.

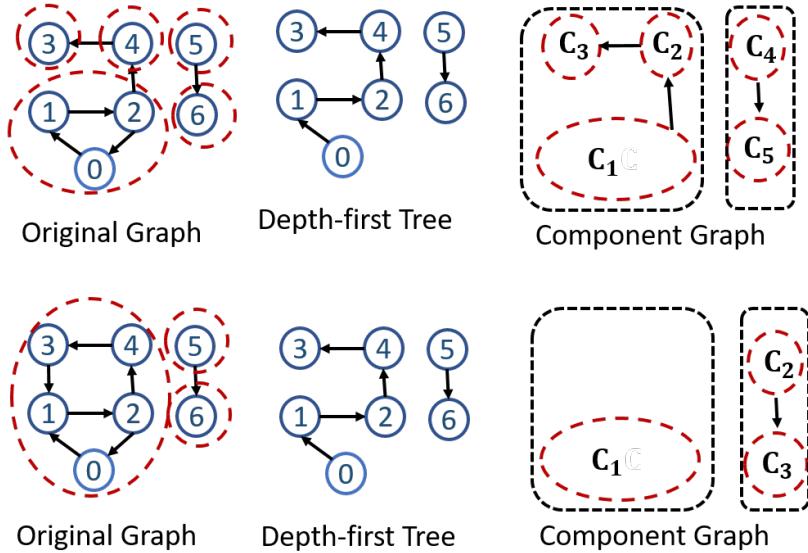


Figure 19.5: DF tree is a superset of SCCs. The black dashed block marks a df-tree.

Search Tree is NOT Enough Back to the algorithms to find SCCs in the directed graph. We can start from the concept of Depth-first Tree we get starts from a source vertex u . As shown in Fig. 19.5, we draw the depth-first tree of each graph, and we would find that a depth-first tree is actually a set of SCCs. The two graphs have different component graphs yet they share the same depth-first tree. This is due to the fact a depth-first tree guarantee to find connection of vertices in one direction, therefore, it connects SCCs and we are sure these SCCs are connected through one direction. We draw the conclusion that depth-first trees are not enough for us to find SCCs in DAG but they can provide crucial clues in the process.

Separate SCCs from Each Depth-first Tree

Step 1: Topological Sort of Component Graph Inspired by the above fact that depth-first tree connect a set of SCCs in one direction, and if we look each SCCs as a single vertex as in the component graph, we can do a topological sort and we get vertices in order. This process is exactly the same as the topological sort in Section 19.2. Thus, here we skip the code in

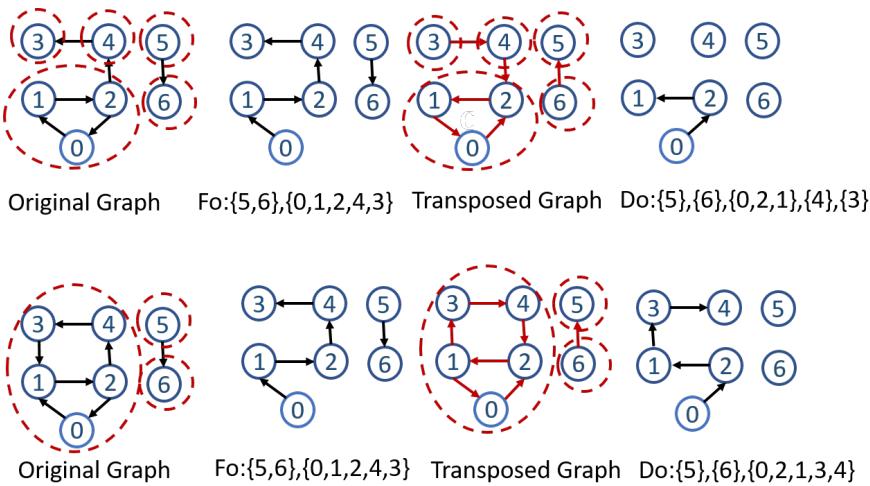


Figure 19.6: The process of Kosaraju's SCC finding algorithm

this section. And the ordering of all vertices in original graph is shown in the second column of Fig. 19.6.

Step 2: Transposed Graph Here, we have a new concept, the transpose of the original graph G as G^T , where $G^T = (V, E^T)$ and $E^T = \{(u, v) : (v, u) \in E\}$. Here we draw the transpose graph of the original graph in Fig. 19.6. In the transpose graph, each SCC is still a SCC and within it each pair of vertices is connected with each other. Between each SCC, the directed edge is reversed.

Step 3: Depth-first Tree on Transpose Graph If we apply depth-first search on the transposed graph with the topological ordering of vertices, each depth-first tree will be a SCC. Assuming two SCCs, C and C' and C is predecessor of C' . Then we find C 's depth-first tree earlier than C' , and since the reversed edges, C will never have edges to other SCCs. Therefore, another depth-first search over all vertices with the topological ordering, we will find the SCCs. And we draw the DAG of component graph by reversing the edge between different SCC.

The SCCs algorithm we learned is **Kosaraju's algorithm**. Try to check out Tarjan's

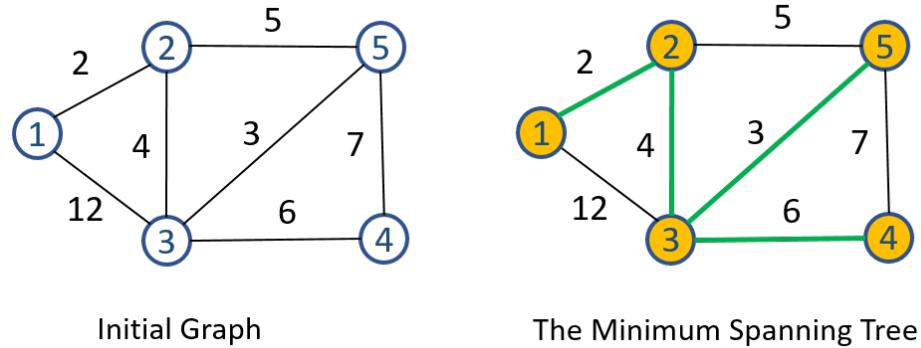


Figure 19.7: Example of minimum spanning tree in undirected graph, the green edges are edges of the tree, and the yellow filled vertices are vertices of MST.

19.4 Minimum Spanning Trees

A spanning tree of an undirected graph $G = (V, E)$ in the field of graph theory is defined as a subgraph (V, SE) that is a tree which includes all vertices of G , with minimum possible number of edges. In order for a graph to have spanning tree, the graph should be connected. Otherwise, we use spanning forests to refer instead. If the undirected graph G is edge-weighted, a minimum spanning tree (MST) or minimum weight spanning tree is a subset of edges that connects all the vertices together, without any cycles and with the minimum possible accumulated edge weights.

In graph related applications, we most likely need to first need to run connected component test to make sure the graph's connectivity. Afterwards, if the graph is weighted, finding minimum spanning tree (MST) or minimum weight spanning tree can be cost saving. We show an example of MST in Fig. 19.7.

Before we move on to actual algorithms for MST, we introduce more related terminologies that smooth the communication.

Definition of Cut, Cross Edge and Light Edge A cut denoted as $(S, V - S)$ of an undirected graph is a partition of V . As shown in Fig. 19.8 a cut is denoted with red curve. Here this cut separate V into $S = \{1, 2, 3\}$ and $V - S = \{4, 5\}$. A **cross edge** is defined on the basis of a cut, it is defined as an edge crosses the cut if one of its endpoint is in S and the other is in $V - S$. In this figure, the cross edges include $(2, 5)$, $(3, 5)$, and $(3, 4)$ which are emphasized with bold edges. A **light edge** is the minimum edge among all cross edges. We also say a cut **respects** a set A of edges if no edge in A crosses the cut, which here are

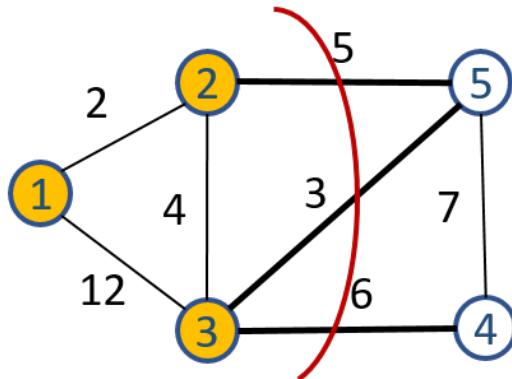


Figure 19.8: A cut denoted with red curve partition V into $\{1,2,3\}$ and $\{4,5\}$.

19.4.1 Prim Algorithm

Prim Algorithm is a greedy algorithm which picks a vertex randomly and grow the tree one edge at a time incrementally. Here, our exemplary input undirected graph is shown in Fig. 19.7.

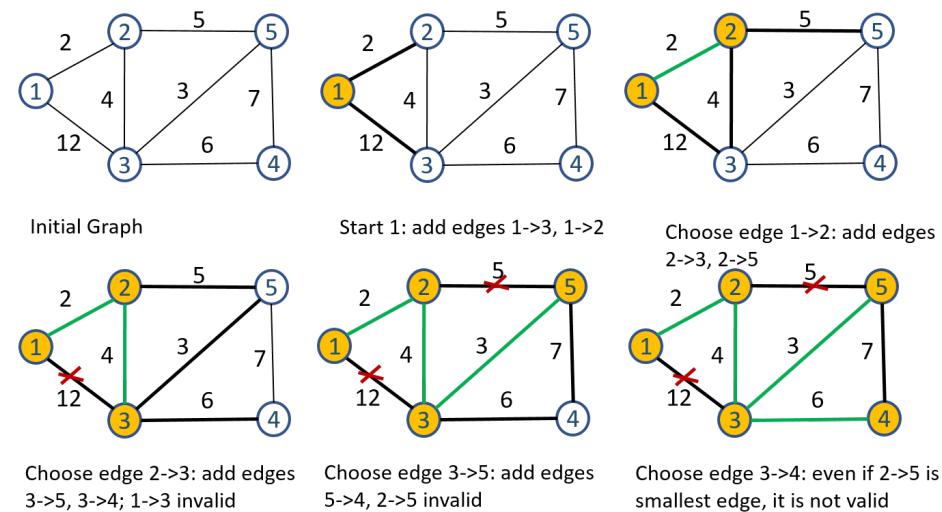


Figure 19.9: Prim's Algorithm

As Fig. 19.9 shows, prim's algorithm's mst always start from an arbitrary root vertex, which is node 1 here and grow until the tree spans all the vertices in V . Each step, it adds to the tree T a light edge that connects T to an isolated vertex. For convenience, we define the tree node set S to track all of mst's current found vertices. And the cross edges set include all cross edges between S and $T - S$, which here we denote is as CE

We first pick a vertex as initial tree vertex set, which here is simply $\{1\}$. First, we find all of its cross edges of the initial vertex, which as we have defined before, a cross edge is an edge that one endpoint is the vertex and the other endpoint is in $T - S$. We add these cross edges to our all edge set. We have edge $CE = \{(1, 3), (1, 4), (1, 2)\}$. Among the cross edges, we find the smallest cross edge from it which is equivalently the light edge $(1, 2)$. We then remove the light edge out of our cross edge set, and add the other endpoint 2 in the tree node set, having $S = \{1, 2\}$ and $CE = \{(1, 3), (1, 4)\}$.

For the next new mst vertex 2, we repeat the above process, adding cross edges $(2, 3), (2, 5)$, we end up with $CE = \{(1, 3), (1, 4), (2, 3), (2, 5)\}$. The current light edge is $(2, 3)$, remove it, thus $S = \{1, 2, 3\}$. To note that after this operation, edge $(1, 3)$ is no longer a cross edge anymore due to the fact that both 1 and 3 are in the set S . Therefore we remove it, $CE = \{(1, 3), (1, 4), (2, 5)\}$. We keep repeating such process: (1) adding cross edges starting with the last most vertex in MST; (2) removing light edge from current CE and adding the other endpoint to S ; to get the exact result shown above.

Clean and Efficient $O(E \log V)$ Implementation with Min-heap

To track cross edges, we customize a data structure: includes two endpoints, `pid` and endpoint id `id`, and the weight of the edge `w`. For simplicity, we use `PriorityQueue` from `queue` module to implement CE . Because the priority of the edges in the set is dependable of its weight— the smaller the higher of the priority, we customize its comparison based magic functions: `__lt__` and `__eq__`.

```

1 class Edge:
2     def __init__(self, pid, id, w):
3         self.pid = pid
4         self.id = id
5         self.w = w
6     def __lt__(self, other):
7         return self.w < other.w
8
9     def __eq__(self, other):
10        return self.w == other.w
11
12    def __str__(self):
13        return str(self.pid) + '→' + str(self.id) + ':' + str(self.w)
14
15    def __repr__(self):
16        return self.__str__()

```

We implement S with a `set` data structure with initial starting vertex. Using a set enables us to check if a vertex is in the MST or not in constant time. Each step to pick the light edge, we do not delete the unqualified

edges out as time goes by, instead when we pop out a potential light edge, we check its validity by validating if both of its endpoints have already been in the set S . Thus the algorithm is implemented as follows:

```

1 import queue
2 def prim(g, n):
3     # step 1:
4     start = 1
5     V = {start} #spanning tree set
6     E = queue.PriorityQueue() # the set of all edges,
7     ans = []
8
9     while len(V) < n:
10         # add edges of start , and the other endpoint is in nv
11         idlst = g[start]
12         for id, w in idlst:
13             if id not in V:
14                 E.put(Edge(start, id, w))
15
16     while E:
17         # pick the smallest edge
18         minEdge = E.get()
19
20         if minEdge.id not in V:
21             # set the new id as start
22             start = minEdge.id
23             # add this id to the set of tree nodes
24             V.add(minEdge.id)
25             ans.append(minEdge)
26             break
27     return ans

```

Run the above example with input `a= 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)]`, and code `print(prim(a, 5))`:
The output is:

```
[1->2:2, 2->3:4, 3->5:3, 3->4:6]
```

It is important to notice that the premise that knowing the graph is connected is important. If it is not, the while loop `while len(V) < n:` will be run forever.

Complexity Even if in this implementation, there are two `while` loops and one `for` loop, however the complexity is decided by the inner `for` and `while` loop. With these two loops, we are maintaining a min-heap dynamically with a maximum nodes of E . This makes the time complexity of $O(E \log E)$. In a graph, $E < V^2$, therefore, the complexity become $O(E \log V)$.

Efficient $O(V \log V)$ Implementation with Fibonacci Heaps

If $V < E$, this makes it more efficient using $O(V \log V)$ implementation. Before we head off to the most known efficient implementation, let us start with the algorithm demonstration and do a simple naive implementation.

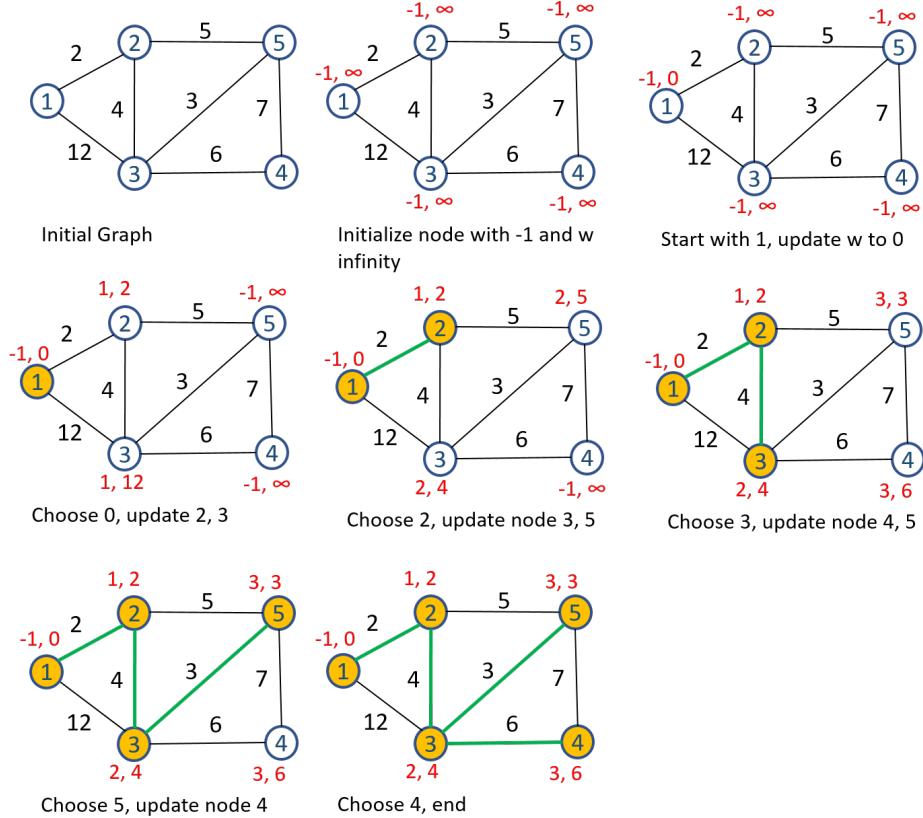


Figure 19.10: Prim's Algorithm

Algorithm Explanation As shown in Fig. 19.10, we first initialize each node's weight to infinity and its parent to -1 . We put these nodes in a data structure. Then, we initialize node start with weight 0. Thus, this node become the node that has the smallest weight. We then update node 1's adjacent valid node's parent and weight if smaller weight is found. The valid node means it is not a vertex already in the Tree node set. Next, we repeat the above process until all nodes has been added to the tree node set.

Algorithm Implementation The data structures that store each node is better to be a PriorityQueue as we have defined in Chapter 10. Here, when we are updating the weight of a node in the queue, we need to save

the new parent id that reaches this vertex. We update the `PriorityQueue` as follows with another item `info` to save any additional information we need.

```

1 from heapq import heappush, heappop, heapify
2 from typing import List
3 import itertools
4 class PriorityQueue:
5     def __init__(self, items: List[List] = []):
6         self.pq = []                                     # list of entries
7         self.entry_finder = {}                          # mapping of tasks to
8         entries
9         self.REMOVED = '<removed-task>'           # placeholder for a
10        removed task
11        self.counter = itertools.count()            # unique sequence
12        count
13        # add count to items
14        for p, t, info in items:
15            item = [p, next(self.counter), t, info]
16            self.entry_finder[t] = item
17            self.pq.append(item)
18            heapify(self.pq)
19
20    def add_task(self, task, priority=0, info=None):
21        'Add a new task or update the priority of an existing task
22        '
23        if task in self.entry_finder:
24            self.remove_task(task)
25        count = next(self.counter)
26        entry = [priority, count, task, info]
27        self.entry_finder[task] = entry
28        heappush(self.pq, entry)
29
30    def remove_task(self, task, info=None):
31        'Mark an existing task as REMOVED. Raise KeyError if not
32        found.'
33        entry = self.entry_finder.pop(task)
34        entry[-2] = self.REMOVED
35
36    def pop_task(self):
37        'Remove and return the lowest priority task. Raise
38        KeyError if empty.'
39        while self.pq:
40            priority, count, task, info = heappop(self.pq)
41            if task is not self.REMOVED:
42                del self.entry_finder[task]
43                return task, info, priority
44        raise KeyError('pop from an empty priority queue')

```

Even since the `PriorityQueue` is implemented, the implementation of Prim's algorithm based on the weight of each vertex will be:

```

1 def primMst(g, n):

```

```

2 # initialization
3 q = PriorityQueue()
4 S = []
5 ans = []
6 for i in range(n):
7     q.add_task(task=i+1, priority=float('inf'), info=None)
8 q.add_task(1, 0, info=1)
9 S = {1}
10
11 # main process
12 while len(S) < n:
13     minId, info, p = q.pop_task()
14     ans.append((info, minId, p))
15     S.add(minId)
16     for v, w in g[minId]:
17         if v not in S and w < q.entry_finder[v][0]:
18             q.add_task(v, w, minId)
19
20 return ans
21

```

We run function `print(primMst(a, 5))` will get the following output:

```
[(1, 1, 0), (1, 2, 2), (2, 3, 4), (3, 5, 3), (3, 4, 6)]
```

With the priority queue, the difference compared with the previous implementation is that we do not need to update the priority queue for any edge in the graph. Instead, only edge that has smaller weight compared with previous edge that can reach to adjacent nodes of current chosen node.



Can you write a customized minHeap other than standard `heapq` or `PriorityQueue()` to replace `q`? And what is the complexity then?

Here we move on further to implement it with `heapq`. The difference is we might need to update to implement a customized heap with efficient decrease-key function that in $O(\log n)$. Otherwise, the value of MST Vertex set, this step takes $O(V)$

19.4.2 Kruskal's Algorithm

Kruskal's algorithm takes a different approach. It first starts with the vertices set V and view them as a forest and each vertex is a single tree with only root node. The main process of the algorithm is to merge each tree by joining two at a time by choosing a safe edge. A safe edge is the light edge that connects two trees C_1 and C_2 . With our above example, our process of Kruskal's algorithm is shown in Fig. 19.11.

At first, we assign a set id for each vertex which is marked with red color above each vertex. Before we start to merge trees, we sort all edges E in

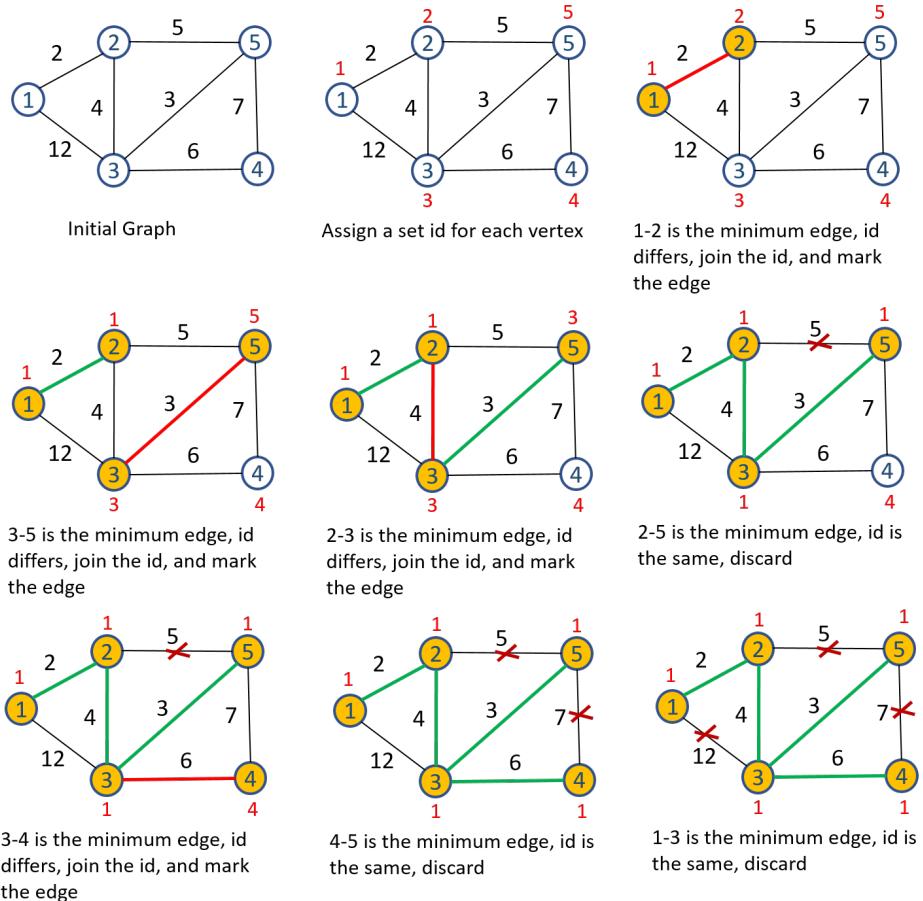


Figure 19.11: The process of Kruskal's Algorithm

increasing order, and they will be $(1, 2)$, $(3, 5)$, $(2, 3)$, $(2, 5)$, $(3, 4)$, $(4, 5)$, $(1, 3)$. At first, edge $(1, 2)$ has the minimum weight as of 2. We merge these two trees by modifying vertex 2's set id to 1 (here we just change it to the smaller one). For edge $(3, 5)$, we merge these two vertices. Same for edge $(2, 3)$. For edge $(2, 5)$, because vertex 2 and 5 are already in the same tree by checking its set id, we just move on. Following this, we will eventually get our edges SE for MST. A simple Python implementation is given as:

```

1 def KruskalMst(g, n):
2     '''Implement Kruskal algorithm using simple dictionary instead
       of disjoint set'''
3     edges = []
4     # BFS traversal
5     visited = set()
6     for i in range(1, n+1):
7         visited.add(i)
        for j, w in g[i]:

```

```

9         if j not in visited:
10            edges.append(Edge(i, j, w))
11    edges = sorted(edges)
12
13 # main process
14 ids = [i for i in range(1, n+1)]
15 node_set = dict(zip(ids, ids))
16 set_node = dict(zip(ids, [[i] for i in range(1, n+1)]))
17 SE = []
18
19 for edge in edges:
20    pid, id = edge.pid, edge.id
21    if node_set[pid] != node_set[id]:
22        #print(pid, sets[pid], id, sets[id])
23        sid, lid = node_set[pid], node_set[id]
24        if lid < sid:
25            sid, lid = lid, sid
26
27        for idx in set_node[lid]:
28            node_set[idx] = sid
29
30        set_node[sid] += set_node[lid]
31        del set_node[lid]
32
33    SE.append(edge)
34 print(node_set, set_node)
35 return SE

```

With `print(KruskalMst(a, 5))`, our output is:

```
{1: 1, 2: 1, 3: 1, 4: 1, 5: 1} {1: [1, 2, 3, 5, 4]}
[1->2:2, 3->5:3, 2->3:4, 3->4:6]
```

19.4.3 Compare Kruskal's to Prim's Algorithm

1. **Global-wise VS Local-wise Greedy:** Kruskal's is a global-wise greedy approach, because it considers the edges in decreasing order of their weight. While, the Prim's algorithm is a local-wise greedy. It adds cross edges gradually into its min-heap.
2. **Forest VS Tree:** In Kruskal's, it creates the MST by merging different trees in the forest choosing the light edge global-wisely. While, the Prim's start with root node of a one node tree, and each time add a light edge among all valid cross edges.
3. **Set VS Min-Heap:** In all, Kruskal's utilize the technique of `set` and sorting, and the Prim's dynamically maintain a `min-heap` to add/remove edges in one easier implementation. At a more difficult implementation, we need to modify the edge's weight in the min-heap. Therefore, in all, in the case of implementation, we can choose one by analyze the relation between $|V|$ and $|E|$.

19.5 Single-Source Shortest Paths

In this section and the next one(Sec. 19.6) we will discuss graph search algorithms that are designed to find shortest paths for a given weighted, directed graph (WDG) $G = (V, E)$, with weight function $w : E \rightarrow R$ mapping edges to real-valued weights. The weight of a path $p = < v_0, v_1, \dots, v_k >$ is the sum of the weights of its constituent edges, and will be denoted as $w(p)$ through this book.

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i) \quad (19.1)$$

We define the shortest-path weight $\sigma(u, v)$ by

$$\sigma(u, v) = \begin{cases} \min\{w(p) : u \rightarrow^p v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases} \quad (19.2)$$

Therefore, a shortest path is from vertex u to vertex v is then defined as any path p with weight $w(p) = \sigma(u, v)$.

Can a shortest path contain a cycle? If the graph contains a negative-weight cycle (summation over one cycle) reachable from source s , then no path from s to a vertex on the cycle can be a shortest path-we can always find a path with lower weight by following the proposed weighted cycle. Therefore, if there is an negative-weight cycle on some path from s to v , we define $\sigma(s, v) = -\infty$.

How about a positive-weight cycle? Assume $p = [v_0, v_1, \dots, v_k]$ is a path and $c = [v_i, v_{i+1}, \dots, v_j]$ is a positive-weight cycle on this path (so that $v_i = v_j$ and $w(c) > 0$), then the path p' that deleted the cycle from p , which means $c = [v_i, v_{i+1}, \dots, v_j] \rightarrow < v_i >$, our resulting path $p' = < v_0, v_1, \dots, v_i, v_{j+1}, \dots, v_k >$ has weight $w(p') = w(p) - w(c) < w(p)$. Thus the original p that with the positive-weight cycle can not be a shortest-path. For example, in Fig. ??.

Therefore, when we are designing shortest-path algorithms for our given problems, it is important to address questions:

1. Does the graph have a weight function that is non-negative? If true, we can use **Dijkstra's algorithm** detailed in Subsection 19.5.2.
2. If not, is the graph acyclic? If it is acyclic, then a negative-weight cycle will never exist in the graph. We solve our problems with **Single-source shortest paths in directed acyclic graphs** in Subsection 19.5.3.
3. If the graph is potentially cyclic, and potentially end up with negative-weight cycle, we go for **Bellman-Ford Algorithm** (subsection 19.5.1) for answer.

Combinatorial Optimization Shortest path problem is a truly *combinatorial optimization problem*. An exhaustive search with $O(b^d)$ time complexity is not tractable. As we have learned from Part VI, dynamic programming and greedy algorithms are often the strategies to resolve optimization problems. So, in this Chapter, just to keep these two strategies in mind too.

Variants of Shortest-path Problems In the shortest-path problems, there are concepts like sources and targets which are all vertices in a graph. Categorized by the relation between sources and targets, we have the following four variants in terms of shortest-path problems:

1. **Single-source shortest-paths problem:** Find a shortest path from a given source s vertex to each vertex $v \in V$.
2. **Single-target shortest-paths problem:** 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 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. Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.
4. **All-pairs shortest-paths 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 the next section (Sec. 19.6).

As we can see, the first type *single-source shortest-paths problem* is fundamental to any other types in the shortest-paths world. We dedicate the first section to address the single-source shortest-paths problem.

Naive Solution A naive solution to get the shortest path is simply do tree-search starts from a given source vertex s within the graph within path length of $|V| - 1$. In the process, we track the minimum weight of each target vertices, and return the minimum weight. Because we limit the maximum length to be $|V| - 1$, the algorithm is guaranteed to stop. This can be implemented with either BFS or DFS traversal as you prefer. This works for graph having negative weights, but not with negative cycles. This gives us a polynomial complexity as of $O(b^{|V|})$, where b is the maximum branch of a vertex. If the graph is dense, which means each branch can be as large as $|V|$. This naive solution can have a complexity of $|V|^{|V|}$.

We now start to observe the shortest path problem as any optimization problem and discover its better solution by observing its properties. First,

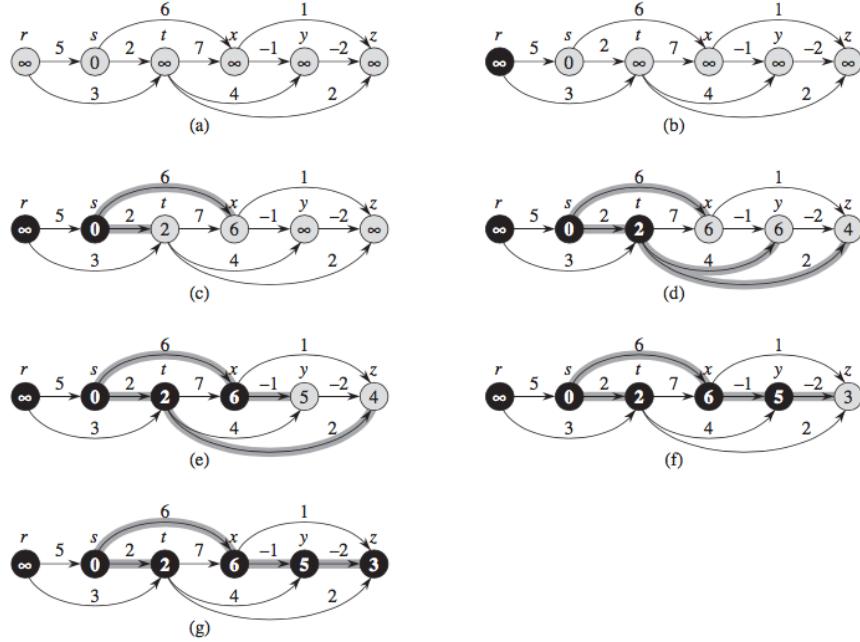


Figure 19.12: Flattening graph for better observation

let us first flatten the graph as shown in Fig. 19.12.

In the shortest-path, the size of the problem is defined as the path length. This gives us a hint that we can use divide and conquer to solve our larger problem on the basis of smaller problems.

Dynamic Programming

Now, our problem is to find $\sigma(s, i)$, it can be defined as the minimum weight of any path between $s \rightarrow i$ that contains at most $|V| - 1$ edges. And we define l_i^m be the minimum weight of any path from s to i that contains at most m edges, thus,

$$\sigma(s, i) = l_i^{|V|-1}. \quad (19.3)$$

We further divide the problem of getting $\sigma(s, i)$ into $|V| - 1$ possible subproblems $l_i^m, m \in [0, |V| - 1]$. Just like the subproblem property we learned in Dynamic Programming chapter, the result of a larger subproblem relies on smaller subproblems. This relation is decided by Recurrence Function.

Now, l is our dynamic programming vector, where it needs to be initialized.

Initialization When $m = 0$, there is a shortest path from s to i with no edges if and only if $s = i$.

$$l_i^0 = \begin{cases} 0 & \text{if } s = i \\ \infty & \text{otherwise} \end{cases} \quad (19.4)$$

Recurrence Function Because of the optimal substructure property, When $m \leq 1$, just like reduction, and the **recurrence relation function** between the subproblem is

$$l_i^{(m)} = \min(l_i^{(m-1)}, \min_{k \in [1,n]} (l_k^{(m-1)} + w_{ki})) \quad (19.5)$$

The second part min is the process of looking at all predecessors of i . When $k = j$, $w_{ii} = 0$, thus it can be simplified as

$$l_i^{(m)} = \min_{k \in [1,n]} (l_k^{(m-1)} + w_{ki}) \quad (19.6)$$

The implementation of the above equation depends on the graph representation. k and i is to enumerate all possible edges. If it is more compressed like with adjacency list, this means we relax all edges $|E|$, while if it is adjacency matrix, we need two for loop with $|V|^2$.

With the optimal substructure property, we developed the recurrence function to solve the shortest path problems using fundamental dynamic programming method.

Optimal Substructure Property

In our case, given V vertices, the maximum path-length will be $V - 1$, and smallest subproblem is 0-length with just one source vertex. Let's say our problem here is to obtain $\sigma(s, v_i)$ for each $v_i \in V$. Say if we get the shortest path between s and v , and on this path we know the predecessor of v which is vertex u , then we finds the shortest path between s and u too. This is the optimal substructure property:

$$\sigma(s, v) = \sigma(s, u) + w(u, v) \quad (19.7)$$

We prove the optimal substructure property with: First, u and v is connected and u is predecessor of v on the shortest path of $s \rightarrow v$, we define $p_l(s \rightarrow u)$ as the path from s to u with length l .

$$\sigma(s, v) = \min_{l=0}^{|V|-1} w(p_l(s \rightarrow v)) \quad (19.8)$$

$$= \min_{l=0}^{|V|-1} w(p_l(s \rightarrow u \rightarrow v)) \quad (19.9)$$

$$= \min_{l=0}^{|V|-1} \{w(p_l(s \rightarrow u)) + w(u, v)\} \quad (19.10)$$

$$= \sigma(s, u) + w(u, v) \quad (19.11)$$

Now, consider all of v 's predecessor, for v we have u_0, u_1, u_2, \dots . And u is the shortest:

$$\sigma(s, v) = \min_i \sigma(s, u_i) + w(u_i, v). \quad (19.12)$$

In this section, we shall see how we develop two shortest path algorithms based on the above two properties: Bellman-Ford Algorithm uses the dynamic programming, and the shortest path algorithms uses the optimal substructure properties.

19.5.1 The Bellman-Ford Algorithm in General

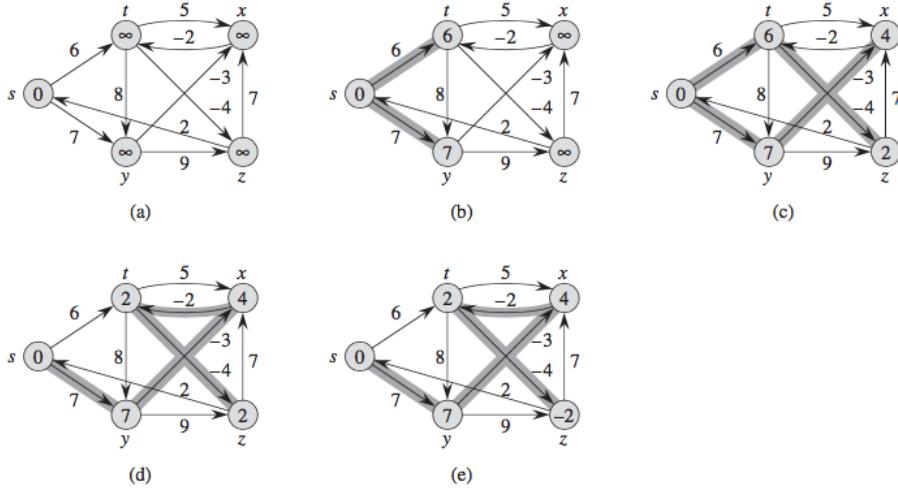


Figure 19.13: The execution of Bellman-Ford's Algorithm

In this section, we will demonstrate our algorithm on the graph shown in Fig. 19.13, and we represent this graph as follows in Python with a dictionary. Such representation is more natural and clear in real production. However, when we are implementing our algorithms, in order to represent l and π vector, we assign $[0, |V| - 1]$ to each vertex in a `label_map`.

```

1 g = {
2     's': [( 't', 6), ( 'y', 7)],
3     't': [( 'x', 5), ( 'y', 8), ( 'z', -4)],
4     'x': [( 't', -2)],
5     'y': [( 'x', -3), ( 'z', 9)],
6     'z': [( 'x', 7)]
7
8 }
```

Bellman-Ford algorithm is invented on the dynamic programming we developed. Let's see this property is applied.

Initialization The first step in which we initialize the weight of $p(s, s) = 0$, and all other vertices will have $p(s, v_i) = \infty$.

Recurrence Function and Result Format From recurrence function shown in Eq. 19.6 and ???. We know we need to have $|v| - 1$ passes to obtain all shortest path weight to each vertex. For example, as shown in Fig. 19.13. This recurrence function gives us clue to run $V - 1$ passes of BFS with relaxation on the weight of the edges. We can observe that in the first pass, we get the shortest-path $w(s, t) = 6$. At the third path with path-length of 3, its shortest-path weight decrease again due to the addition of negative weighted path and become 2.

The Python code is shown:

```

1 import random
2 def bellman_ford(g, s):
3     # get a label map
4     s = 's'
5     n = len(g)
6     print(n, g.keys())
7     label_map = dict(zip(g.keys(), [i for i in range(n)]))
8     print(label_map)
9     si = label_map[s]
10    # assign l and pi vector
11    l = [0 if i==si else sys.maxsize for i in range(n)]
12
13    keys = list(g.keys())
14    #random.shuffle(keys)
15    print('10: ', 1)
16    for p in range(n-1): #n-1 passes
17        # relax all edges
18        for k in keys:
19            ki = label_map[k]
20            for i, w in g[k]:
21                ii = label_map[i]
22                l[ii] = min(l[ii], l[ki]+w)
23    print('l{}: {}'.format(p+1, l))

```

The output of running `bellman_ford(g, 's')` will be:

```

1 10: [0, 9223372036854775807, 9223372036854775807,
      9223372036854775807, 9223372036854775807]
2 11: [0, 6, 4, 7, 2]
3 12: [0, 2, 4, 7, 2]
4 13: [0, 2, 4, 7, -2]
5 14: [0, 2, 4, 7, -2]

```

And, we would see different intermediate result if we reordering the relaxing ordering of edges. But, eventually, the result of the last pass will end up being the same no matter what ordering.

Complexity Analysis It is easy to see that the complexity of Bellman-Ford is $O(|V||E|)$ with $O(|V|)$ passes and each pass with relax over $|E|$

edges.

19.5.2 Dijkstra's Algorithm in Non-negative Weighted DG

When our given graph is a non-negative weighted direct graph and the problem is to find all shortest-paths from source vertex s . Under this configuration, our problem has one more nice property on the basis of the **optimal substructure property**, which we call it **Non-decreasing Path Property**. This property states as we extend a path $p = [v_0, v_1, \dots, v_k]$, and that the resulting path weight will be larger or equal when the extended edges have all 0 weight.

The main process of dijkstra is simple and clear, however it needs the support of a priority queue which can support operation like item lookup and updation.

```

1 from heapq import heappush, heappop, heapify
2 from typing import List
3 import itertools
4 class PriorityQueue:
5     def __init__(self, items: List[List] = []):
6         self.pq = []                                     # list of entries
7         arranged in a heap
8         self.entry_finder = {}                         # mapping of tasks to
9         entries
10        self.REMOVED = '<removed-task>'           # placeholder for a
11        removed task
12        self._counter = itertools.count()            # unique sequence
13        count, this is hidden from user
14        # add count to items
15        for p, t, info in items:
16            item = [p, next(self._counter), t, info]
17            self.entry_finder[t] = item
18            self.pq.append(item)
19            heapify(self.pq)
20
21    def add_task(self, task, priority=0, info=None): # O(logE)
22        'Add a new task or update the priority of an existing task
23        '
24        'the old task is removed from entry_finder but still
25        remained in the heap'
26        if task in self.entry_finder:
27            self._remove_task(task)
28        count = next(self._counter)
29        entry = [priority, count, task, info]
30        self.entry_finder[task] = entry
31        heappush(self.pq, entry)
32
33    def _remove_task(self, task, info=None):# O(1)
34        'Mark an existing task as REMOVED. Raise KeyError if not
35        found.'

```

```

29     entry = self.entry_finder.pop(task)
30     entry[-2] = self.REMOVED
31
32     def pop_task(self): #O(logE)
33         'Remove and return the lowest priority task. Raise
34         KeyError if empty.'
35         while self.pq:
36             priority, count, task, info = heappop(self.pq)
37             if task is not self.REMOVED:
38                 del self.entry_finder[task]
39                 return task, info, priority
40             raise KeyError('pop from an empty priority queue')
41
42     def get_task(self, taskid):
43         '''return task information given task id'''
44         if taskid in self.entry_finder:
45             p, _, t, info = self.entry_finder[taskid]
46             return p, info
47         else:
48             return None, None
49
50     def empty(self):
51         return not self.entry_finder

```

Mutant BFS Let's refresh how we use BFS to find the shortest path, where all edges have the same weight=1. The idea of Dijkstra's Algorithm is highly similar to this. In the simplified version, the FIFO queue manages to sort vertices in increasing order with the shortest path length as weight. The difference here is: First, the FIFO queue needs be replaced by a min-heap priority queue; Second, the weight of the vertices in the queue needs to be updated if a smaller weight is found in the expansion. We give the BFS-like Dijkstra's Algorithm implementation:

```

1 def dijkstra_bfs(g, s):
2     '''implement dijkstra with BFS style'''
3     Q = PriorityQueue()
4     S = []
5     Q.add_task(task=s, priority=0, info=None) #add source node
6     visited = set()
7     while not Q.empty():
8         min_id, min_w, p = extract_min(Q)
9         visited.add(min_id)
10
11        S.append((min_id, min_w, p))
12
13        for id, w in g[min_id]: #target
14            if id in visited: #already found the shortest path for
15                continue
16            pw, pp = Q.get_task(id)

```

```

17     # first time to add
18     if not pw or not pp:
19         Q.add_task(task=id, priority=min_w+w, info=min_id)
20         continue
21
22     # if its in the queue, update its weight
23     new_w, new_p = w, pp
24     if min_w + w < pw:
25         new_w = min_w + w
26         new_p = cid
27     Q.add_task(task=id, priority=new_w, info=new_p)
28 return S

```

Based on the optimal substructure property and with the non-decreasing path property, we can make the Bellman Ford Algorithm “greedy”. At each step, dynamic programming uses Eq. 19.6 to update the vector $l_i^{(m)}$ by trying all possible edges that extends the paths between s and i by one more length. This is equivalent to relax all edges in the graph. Second, it can only get the optimal solution at the very end of running all passes. However, at the greedy version, it decides the optimal solution with “only” local information and each step it only relax vertices that are reachable from the previous minimum vertex. This greedy version of Bellman Ford algorithm is called *Dijkstra’s Algorithm*

Dijkstra’s Algorithm To be specific, we define a set S of vertices whose final shortest-path has already been determined. And $V - S$ is the other set. Dijkstra still has $|V| - 1$ passes, and at each pass, it only makes local decision and local updates by repeatedly selecting the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S , and relaxes all edges leaving u . It can be implemented with a min-priority queue Q if vertices. We can see our high level code shown in:

Here, we show the main process of this algorithm in Fig. 19.14. ‘s’ is the source vertex and is located at the leftmost. At first step is the initialization, where we set the weight of all vertices except the source vertex to ∞ and assign 0 to the source. At step b, we relax edges leaving source s , and adds y to our set S . At step c. we relax edges leaving the last minimum vertex y , and get to add z to S . At step d and e it relaxes edges leaving z and t . Step f is our last step. By running the example, and at the last step, we can clearly see we found the shortest-path tree.

Correctness of Dijkstra’s Algorithm To prove the correctness of a greedy algorithm is not easy matter, and often the contradiction method can be used. In this process, we need to show that for each vertex $u \in V$, we have $u.d = \sigma(s, u)$ at the time when u is added to set S .

1. Initialization: First, $S = \emptyset$, the conclusion holds true.

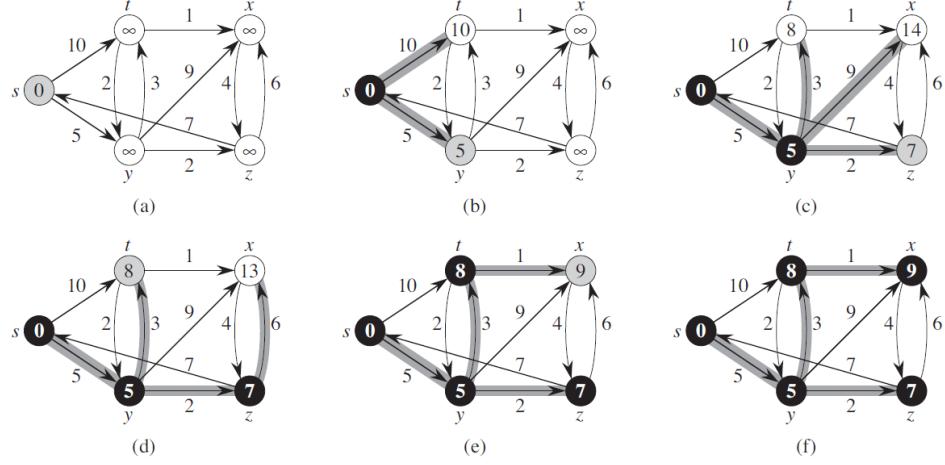


Figure 19.14: The process of Dijkstra's Algorithm. At step f, we get the shortest-path tree.

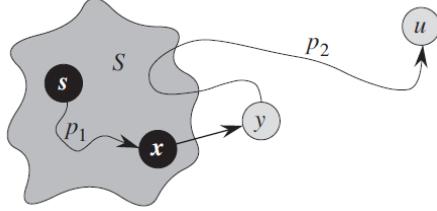


Figure 19.15: The proof of correctness of Dijkstra's Algorithm

2. Maintenance: We wish to show that in each iteration, $u.d = \sigma(s, u)$ for the vertex added to set S . We use contradiction, let u be the first vertex for which $u.d \neq \sigma(s, u)$. For this to work, u cant be s and it also needs to be a vertex that is reachable from s . Importantly, if $u.d \neq \sigma(s, u)$, then there shall exist another path, namely $s \rightarrow x \rightarrow y \rightarrow u$ is shorter. We set the context of one step before, that prior to adding u to S , path p connects s to a vertex x in $V - S$, namely x in $V - S$. At this step, because of induction, $p_1 = s \rightarrow x$ is $\sigma(s, x)$. Edge (x, y) is relaxed at that time.

First, because y appears before u on the shortest-path on shortest path $s \rightarrow u$, and all edges are non-negative, we have:

$$\sigma(s, y) \leq \sigma(s, u), \quad (19.13)$$

$$y.d = \sigma(s, y) \quad (19.14)$$

$$y.d \leq u.d \quad (19.15)$$

Because, both vertices u and y were in $V - S$ when u was added, we

have $u.d \leq y.d$ since we chose u instead. Thus we get an equality $y.d = \sigma(s, y) = \sigma(s, u) = u.d$

There is another version reasoned with non-decreasing property:

1. At the first pass, we start from the source vertex, which is a shortest path itself with weight 0. The next we do it to relax edges that out of the source vertex, and among them, we choose the shortest path. Because of the non-decreasing property, that if v_i^* is the minimum at this step, then all other possible paths that reaches to v_i^* through $S_{v_l} - v_i^*$ will have longer length, therefore weight will only increase, such that it will never come back to harness the current decision.
2. At the second step, we follow the same rule, relax the edges out of v_i^* . Then we have a selection of paths are one and two length away. we would be able to finalize the shortest path from s to one vertex v_i^* by comparing the minimum weights of all of the reachable vertices.

, Once we make the decision, we do not need to relax weights through the whole vertex set other than relaxing the weights of reachable vertices through the current smallest vertex. This is due to the optimal substructure property.

For example, in Fig. ??, shown in (b) at the first pass, we can decide the shortest-path between s and y . The next pass, we only relax the path weights of t, x, z that is reachable through t . Now, we have information of the path weights for those paths that have maximum length of 2. Then vertex z 's shortest path will be finalized. And so on.

Detailed Implementation

Main Functions Functions that used to support the Dijkstra's algorithm.

```

1  from typing import List, Tuple
2
3  def initialize(Q, g, s):
4      # put all edges into q
5      for k in g.keys():
6          Q.add_task(task=k, priority=sys.maxsize, info=None) #task
6          id is the id, info is the predecessor
7          # set weight for vertices one edge away from source
8          Q.add_task(task=s, priority=0, info=None)
9          for id, w in g[s]:
10              Q.add_task(task=id, priority=w, info=s) # weight, id,
11              predecessor
12
13  def extract_min(Q: List[Tuple]):
14      '''extra minimum vertex with the weight using priority queue
14      '''
15      task, info, pri = Q.pop_task()
```

```

16     return task, pri, info
17
18 def update(Q, g, cid, cw):
19     '''current id, w, p'''
20     for id, w in g[cid]: #target
21         pw, pp = Q.get_task(id)
22         if not pw and not pp: # already found the shortest path for
23             this id
24             continue
25         new_w, new_p = w, pp
26         #print(cw, w, pw)
27         if cw + w < pw:
28             new_w = cw + w
29             new_p = cid
30             Q.add_task(task=id, priority=new_w, info=new_p)

```

Dijkstra The main process of Dijkstra's algorithm.

```

1 def dijkstra(g, s):
2     """
3     Q = PriorityQueue() # the set of all edges,
4     S = []
5     # initialize
6     initialize(Q, g, s)
7     while not Q.empty():
8         min_id, min_w, p = extract_min(Q)
9         print(min_id, min_w, p)
10        S.append((min_id, min_w, p))
11        # need to update weight in the queue
12        update(Q, g, min_id, min_w)
13    return S

```

Now, we run experiment with input as `S=dijkstra(g, 's')`, and the output is:

```

1 [( 's' , 0, None) , ( 'y' , 5, 's') , ( 'z' , 7, 'y') , ( 't' , 8, 'y') , ( 'x' , 9, 't')]

```

If we change 's' to 't' with the same weight as of 's' to 'y'(5). The algorithm still works and manage to find the shortest paths.

Compare with Prim's MST Similarly to Prim's algorithm, whose target is to grow a MST starting with a randomly picked vertex, Dijkstra's algorithm is greedy and it will grow a shortest-paths tree one edge at a time starting from the given source. They are also both similar with the implementation using a customized priority queue. It also resembles breath-first search algorithm in that set S corresponds to the set of black vertices in a BFS.

Complexity Analysis There are mainly three operations: `initialize`, `extract_min`, and `update`. The initialize takes $O(V)$ for V vertices, and

assume `extract_min` and `update` each takes x and y . The total time is $O(V + V(x + y)) = O(V(x + y))$. Thus the total time relies on the specific implementation of the priority queue. In our implementation, we used a customized `heapq` which takes V to build the heap, $\log V$ to attract the minimum each time. Because within our implementation, we does not really remove the task from the heap but instead to mark it as “removed”. So we can end up having maximum of E items in the `heapq` this makes be $x = O(\log E)$. For the update, the main cost is `heappush`, here it will be $O(\log E)$. Thus, it makes our implementation of Dijkstra’s Algorithm with a cost of $O(V \log E)$.

19.5.3 General Algorithm in DAG

19.6 All-Pairs Shortest Paths

In this section, we discuss the problem of finding shortest paths between all pairs of vertices in a graph, and the return result is a table of distance between all pairs of vertices.

Weight and Density of Graph Matters Obviously, we can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm $|V|$ times, once for each vertex as the source and relax the distance from different source. If all edges are nonnegative, we can use Dijkstra’s algorithm which is greedy and we can end up with $O(V^3 + VE)$ using min-priority implementation, and $O(VE \log V)$ with min-heap implementation which is an improvement if the graph is sparse.

If the graph has negative-weight edges, instead we can only run Bellman-Ford algorithm once from each vertex as source. The running time will be $O(V^2E)$, which on a dense graph will be $O(V^4)$. In this section, we explore algorithms that can be better in a dense and non-negative graph.

Section Premise In this section, for convenience, most likely we need to represent our graph with $V \times V$ adjacency matrix $W = (w_{ij})$ representing the edge weights of an V -vertex directed graph $G = (V, E)$, that is

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of directed edge } (i, j) & \text{if } i \neq j, \text{ and } (i, j) \in E \\ \infty & \text{if } i \neq j, \text{ and } (i, j) \notin E \end{cases} \quad (19.16)$$

The tabular output of all-pairs shortest paths algorithms is an $V \times V$ matrix too defined as $D = (d_{ij})$, where entry d_{ij} contains the weight of a shortest path from vertex i to vertex j , that is $d_{ij} = \sigma(i, j)$ at termination.

In order to retrieve the shortest path $p = < v_i, \dots, v_j >$, we define **pre-decessor matrix** $\Pi = (\pi_{ij})$, and π_{ij} will be NULL if either $i = j$ or there

is no path from i to j , and otherwise π_{ij} is the predecessor of j on some shortest path from i . we define π_{ij} as:

$$\pi_{ij} = \begin{cases} \text{NULL} & \text{if } i = j \text{ or } \sigma(i, j) = \infty \\ \text{a predecessor of } j \text{ from } i & \text{otherwise} \end{cases} \quad (19.17)$$

What will learn? Just as in previous section the predecessor subgraph G_π forms a shortest-paths tree from a given source vertex, the subgraph induced by the i -th row of the Π matrix should be a shortest-paths tree with root i . The standard All Pair Shortest Path algorithms such as matrix multiplication like shortest paths algorithm and Floyd–Warshall algorithm are just like Bellman–Ford, and are also application of Dynamic Programming.

Dynamic Programming Extension

We extend Eq. 19.7 to two-dimension, our optimal solution structure become:

$$\sigma(i, j) = l_{ij}^{|V|-1} \quad (19.18)$$

And we define l_{ij}^m be the minimum weight of any path from vertex i to j that contains at most m edges. We can easily extend Eq. 19.4 and 19.6 as follows:

$$l_{ij}^0 = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases} \quad (19.19)$$

$$l_{ij}^{(m)} = \min_{k \in [1, n]} (l_{ik}^{(m-1)} + w_{ki}) \quad (19.20)$$

For n nodes in a graph, if there exists no negative-weight cycles, then for every pair of vertices there is a shortest path that is simple and thus contains at most $n - 1$ paths. Therefore, we end up having $\sigma(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)}$.

Range Type Dynamic Programming

This content of this section is highly related to the range type dynamic programming subtype problem patterns shown in Dynamic Programming Special.

Subproblems In the two-dimensional, our solution is a matrix $\sigma(i, j)$, this gives us $n \times n$ subproblems to solve. Here, we define our dp structure differently: $d_{ij}^{(k)}$ is the the minimum path weight in path $i \rightarrow j$ with intermediate vertices drawn from set $\{0, 1, 2, \dots, k - 1\}$.

19.6.1 Bellman-Ford Extension: Matrix Multiplication like Shortest Path Algorithm

In this section, we continue using the dynamic programming recursion function from Section 19.5. The algorithm shown in this section is an extension of the dynamic programming from one-dimensional to two-dimensional according to the resulting (a vector in previous section, and a matrix in this section). And we show its similarity with matrix multiplication in the case of with extra outer loop over $|V|$ vertices as source vertex. The benefits from being matrix multiplication like is we can further improve its efficiency using matrix multiplication optimization which is called **Repeated Squaring**.

The structure of the optimal solution

Recursion function to the all-pairs shortest-paths problem

Computing the shortest-path weights bottom-up The algorithm we implement here will be an extension of the Bellman-Ford algorithm. It equates to run $V - 1$ passes to update the recursion solution. The difference is, within each pass, there is one more extra outer loop to enumerate every vertex in the graph as a start vertex. One pass of the extended bellman-ford algorithm is implemented as:

```

1 import copy
2 def initialize(W):
3     n = len(W)
4     # initialize L
5     L = [[sys.maxsize for _ in range(n)] for _ in range(n)]
6     for i in range(n):
7         L[i][i] = 0
8     return L
9
10 def bellman_ford_one_pass(W, L):
11     '''extend one pass bellman ford to all-pairs'''
12     n = len(W)
13     # dp assign new L
14     L1 = initialize(W)
15     for i in range(n): # start point for this pass
16         for j in range(n): #relax Lsj by adding one more edge s->k->j
17             for k in range(n):
18                 L1[i][j] = min(L1[i][j], L[i][k]+W[k][j])
19     return L1

```

It will update L^i to L^{i+1} . What it does is to relax one more edge for each pair in the L matrix. Now, the extended Bellman Ford algorithm will be:

```

1 def extended_bellman_ford(W):
2     '''same as bellman ford'''
3     n = len(W)

```

```

4 # initialize L, first pass
5 L = copy.deepcopy(W)
6 print('L1 : ', L)
7 print()
8 # n-2 passes
9 for i in range(n-2):
10    L = bellman_ford_one_pass(W, L)
11    print('L{}: {}'.format(i+2, L))
12    print()

```

We start with $L^1 = W$, then we have $n - 2$ passes left. Run the example with `extended_bellman_ford(W)`, we will have the following result:

```

1 L1 : [[0 , 3 , 8 , 9223372036854775807, -4], [9223372036854775807,
0 , 9223372036854775807, 1 , 7], [9223372036854775807, 4 , 0 ,
9223372036854775807, 9223372036854775807], [2 ,
9223372036854775807, -5 , 0 , 9223372036854775807],
[9223372036854775807, 9223372036854775807,
9223372036854775807, 6 , 0]]
2
3 L2: [[0 , 3 , 8 , 2 , -4], [3 , 0 , -4 , 1 , 7], [9223372036854775807,
4 , 0 , 5 , 11], [2 , -1 , -5 , 0 , -2], [8 , 9223372036854775807, 1 ,
6 , 0]]
4
5 L3: [[0 , 3 , -3 , 2 , -4], [3 , 0 , -4 , 1 , -1], [7 , 4 , 0 , 5 , 11],
[2 , -1 , -5 , 0 , -2], [8 , 5 , 1 , 6 , 0]]
6
7 L4: [[0 , 1 , -3 , 2 , -4], [3 , 0 , -4 , 1 , -1], [7 , 4 , 0 , 5 , 3],
[2 , -1 , -5 , 0 , -2], [8 , 5 , 1 , 6 , 0]]

```

Improving Running Time

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. 19.20. The initialization is similar to initialize it to all zeros for matrix multiplication. If we use `.` to mark `bellman_ford_one_pass` operation on L and W , we will have the following relations:

$$L^1 = L^0 \cdot W = W, \quad (19.21)$$

$$L^2 = L^1 \cdot W = W^2, \quad (19.22)$$

$$L^3 = L^2 \cdot W = W^3, \quad (19.23)$$

$$\vdots \quad (19.24)$$

$$L^{n-1} = L^{n-2} \cdot W = W^{n-1} \quad (19.25)$$

Repeated Squaring 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 algorithm. Another one we will see in String algorithms.

In our case, we can utilize repeated squaring technique, and it is a simplified version since $L^d = L^{n-1}$, $d \in [n-1, \infty]$. We can compute L^{n-1} with only $\log(n-1)$ round of one pass operation

$$L^1 = W, \tag{19.26}$$

$$L^2 = W \cdot W, \tag{19.27}$$

$$L^4 = W^2 \cdot W^2, \tag{19.28}$$

$$\vdots \tag{19.29}$$

$$(19.30)$$

The above repetition stops when our $d \geq n-1$.

19.6.2 The Floyd-Warshall Algorithm

20

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.

20.1 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,

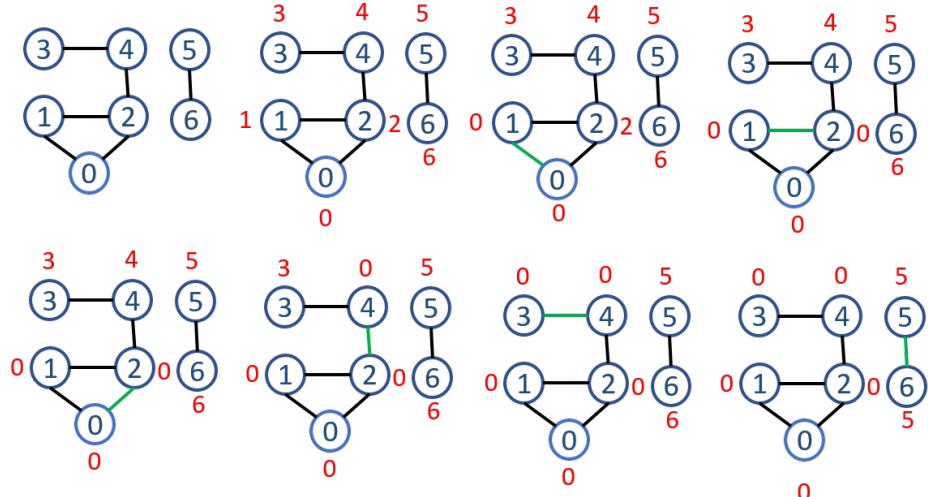


Figure 20.1: The connected components using disjoint set.

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).

¹where new edge will be added and the search based algorithm each time will be rerun to find them again

20.1.1 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).

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
26
27     def union(self, x, y):
28         id_x = self.find_set(x)
29         id_y = self.find_set(y)
30         if id_x == id_y:
31             return
32
33         sid, lid = id_x, id_y
34         if len(self.set_item[id_x]) > len(self.set_item[id_y]):
35             sid, lid = id_y, id_x
36             # merge items in sid to lid
37             for item in self.set_item[sid]:
38                 self.item_set[item] = lid
39             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 hashmap. 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`.

20.1.2 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.

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.

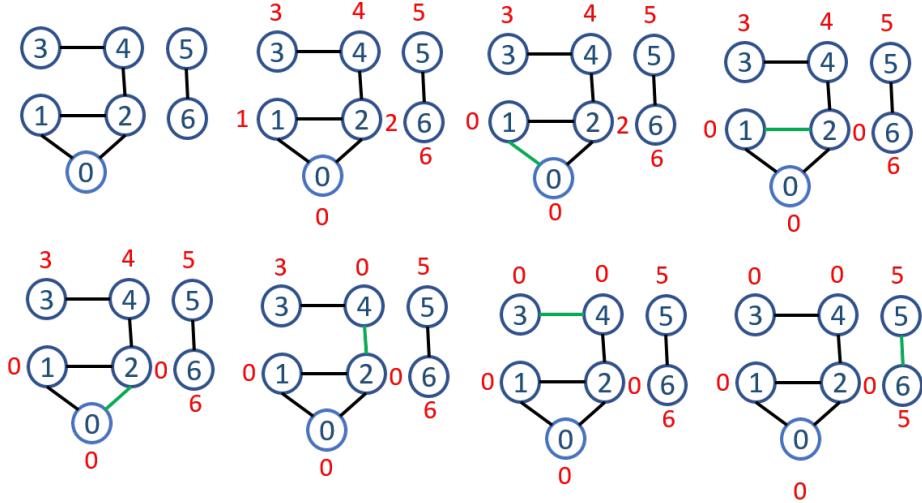


Figure 20.2: A disjoint forest

```

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     def make_set(self, item):
15         '''make set for new incoming set'''
16         if item in self.item_finder:
17             return
18
19         node = Node(item)
20         node.parent = node
21         self.item_finder[item] = node
22         self.sets.add(node)
23         self.n += 1
24
25     def find_set(self, item):
26         # from item->node->parent to set representative
27         if item not in self.item_finder:
28             print('not in the set yet: ', item)
29             return None
30         node = self.item_finder[item]
31         while node.parent != node:
32             node = node.parent
33         return node
34
35     def union(self, x, y):
36         node_x = self.find_set(x)
37         node_y = self.find_set(y)
38         if node_x.item == node_y.item:
39             return
40
41         #the root of one tree to point to the root of the other
42         # merge x to y
43         node_x.parent = node_y
44         #remove one set
45         self.sets.remove(node_x)
46         return
47
48     def __str__(self):
49         ans = ''
50         for root in self.sets:
51             ans += 'set: ' + str(root.item) + '\n'
52         return ans
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?

20.2 Fibonacci Heap

20.3 Exercises

20.3.1 Knowledge Check

20.3.2 Coding Practice

Disjoint Set

1. 305. Number of Islands II (hard)

21

Dynamic Programming Special (15%)

In this Chapter, we categorize dynamic programming into three according to the input data types, including Single Sequence (Section 21.1 and Section 21.2), Coordinate (Section 21.4), and Double Sequence(Section 21.5). 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 21.1: Different Type of Single Sequence Dynamic Programming

Case	Input	Subproblems	$f(n)$	Time	Space
Section 21.1	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n) - > O(1)$
Section 21.2	$O(n)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n)$
Section 21.3	$O(n)$	$O(n^2)$	$O(n)$	$O(n^3)$	$O(n^2)$
Hard	$O(n)$	$O(n^3)$	$O(n)$	$O(n^4)$	$O(n^3)$

Table 21.2: 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:

21.1 Single Sequence $O(n)$

In this section, we will see how to solve the easy type of dynamic programming shown in Table 21.1, 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)

21.1.1 Easy Type

21.1 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)

```

- 21.2 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)

```

- 21.3 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)

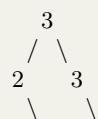
```

21.4 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]



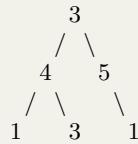
```
3     1
```

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))

```

21.1.2 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.

Introduction to Prefix Sum

In computer science, the prefix sum, cumulative sum, inclusive scan, or simply scan of a sequence of numbers x_0, x_1, x_2, \dots is a second sequence of numbers y_0, y_1, y_2, \dots , the sums of prefixes (running totals) of the input sequence:

```

y0 = x0
y1 = x0 + x1
y2 = x0 + x1+ x2
...

```

For instance, the prefix sums of the natural numbers are the triangular numbers:

input numbers	1	2	3	4	5	6	...
prefix sums	1	3	6	10	15	21	...

Prefix sums are trivial to compute with simple state transfer function $y_i = y_{i-1} + x_i$ with $O(n)$ complexity. 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

```

Despite their ease of computation, prefix sums are a useful primitive in certain algorithms such as counting sort and?? In the following two sections (Sec 21.1.2 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.

Prefix Sum and Kadane's Algorithm Application

21.5 **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 21.3: 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 .
Requesult: 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 21.3.

```

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)

```

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.

21.6 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

```

21.1.3 Subarray or Substring

It will lower the complexity from $O(n^2)$ or $O(n^3)$ to $O(n)$.

21.7 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    print(dp)
24    return max(dp)
25 # input "(()))())()"
26 # 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

```

21.1.4 Exercise

1. 639. Decode Ways II (hard)

21.2 Single Sequence $O(n^2)$

In this section, we will analysis the second type in Table 21.1 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)

21.2.1 Subsequence

21.8 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. 21.1.

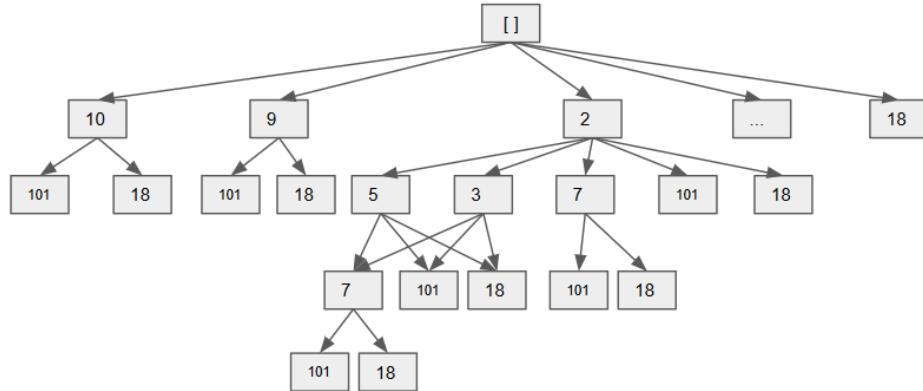


Figure 21.1: State Transfer Tree Structure for LIS, each path represents a possible solution. Each arrow represents a move: find an element in the following elements that's larger than the current node.

```

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,
```

$$f(i) = \begin{cases} 1 + \max(f(j)), & 0 < j < i, arr[j] < arr[i]; \\ 1 & \text{otherwise} \end{cases} \quad (21.1)$$

```

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)
```

21.2.2 Splitting

Need to figure out how to fill out the two-dimensional dp matrix for splitting.

21.9 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 = "leetcode", wordDict = ["leet", "code"]
Output: true
Explanation: Return true because "leetcode" can be
segmented as "leet code".
```

Example 2:

```
Input: s = "applepenapple", wordDict = ["apple", "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 = "catsandog", wordDict = ["cats", "dog", "sand",
"and", "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]

```

花花酱 LeetCode huahualeetcode

139 Word Break

```

wordBreak("leetcode") =
    wordBreak("") && inDict("leetcode")
    || wordBreak("l") && inDict("eetcode")
    || wordBreak("le") && inDict("etcode")
    || wordBreak("lee") && inDict("tcode")
    || wordBreak("leet") && inDict("ode")
    || wordBreak("leetc") && inDict("de")
    || wordBreak("leetco") && inDict("e")
    || wordBreak("leetcod") && inDict("e")

```

```

inDict("leet") = true
inDict("code") = true

wordBreak("") = true
wordBreak("l") = false
wordBreak("le") = false
...
wordBreak("leet") = true
...
wordBreak("leetcode") = true

```

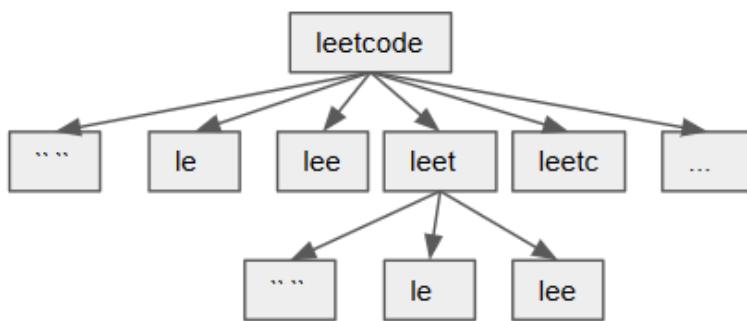


Figure 21.2: Word Break with DFS. For the tree, each arrow means check the word = parent-child and then recursively check the result of child.

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. 21.2 (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, {})

```

- 21.10 **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]

```

- 21.11 **Best Time to Buy and Sell Stock III (L123, 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 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(
30         preProfit ,  postProfit)])
31     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)

21.3 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.

21.3.1 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.

21.12 486. 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.

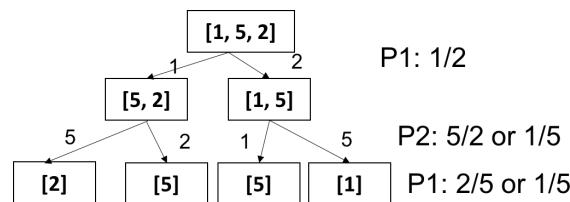


Figure 21.3: Caption

```
1 def PredictTheWinner(nums):
```

```

2
3     """
4     :type nums: List[int]
5     :rtype: bool
6     """
7     if not nums:
8         return False
9     if len(nums)==1:
10        return True
11    #sum[i,j] = sum[j+1]-sum[i]
12    sums = nums[:]
13    for i in range(1, len(nums)):
14        sums[i]+=sums[i-1]
15    sums.insert(0,0)
16
17    dp=[[0 for col in range(len(nums))] for row in
18         range(len(nums)) ]
19    for i in range(len(nums)):
20        dp[i][i] = nums[i]
21
22    for l in range(2, len(nums)+1):
23        for i in range(0, len(nums)-l+1): #start 0, end
24            len-l+1
25            j = i+l-1
26            dp[i][j] = (sums[j+1]-sums[i])-min(dp[i+1][
27                j],dp[i][j-1])
28            n = len(nums)
29    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         """
5         :type nums: List[int]
6         :rtype: bool
7         """
8         n = len(nums)
9         if n == 1 or n%2==0 : return True
10        dp = [[0]*n for _ in range(n)]
11        for l in range(2, len(nums)+1):
12            for i in range(0, len(nums)-l+1): #start 0, end
13                len-l+1
14                j = i+l-1
15                dp[i][j] = max(nums[j] - dp[i][j-1], nums[i]
16                                - dp[i+1][j])
17        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 ] )

```

21.13 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

21.14 312. 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 $\text{nums}[\text{left}] * \text{nums}[i] * \text{nums}[\text{right}]$ 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

$$\begin{aligned}
\text{nums} &= [3, 1, 5, 8] \rightarrow [3, 5, 8] \rightarrow [3, 8] \rightarrow [8] \rightarrow [] \\
\text{coins} &= 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 \\
&= 167
\end{aligned}$$

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 i in range(1, n+1): #length [1,n]
13        for j in range(i, n-1+2): #start [1, n-1+1]
14            j = i+1-1 #end =i+1-1
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        #return from 1 to n
21    return c[1][n]

```

21.15 516. 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 longestPalindromeSubseq(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};

```

21.4 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. 21.4 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 21.4: 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. 21.4.1 and Optimization in Sec. ?? with LeetCode examples to learn how to solve this type of dynamic programming problems.

21.4.1 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 Example 1:
10 Input: m = 3, n = 2
11 Output: 3
12 Explanation:
13 From the top-left corner, there are a total of 3 ways to reach
14   the bottom-right corner:
15 1. Right -> Right -> Down
16 2. Right -> Down -> Right
17 3. Down -> Right -> Right
18 Example 2:
19 Input: m = 7, n = 3
20 Output: 28
21
22
23
24

```

BFS. Fig. 21.4 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:

```

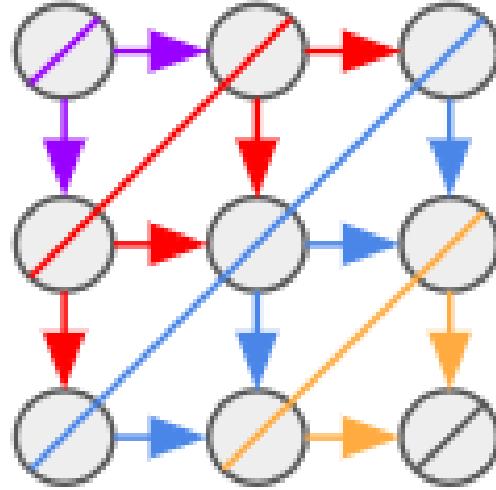


Figure 21.4: One Time Graph Traversal. Different color means different levels of traversal.

```

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 Note that different sequences are counted as different
19 combinations.
20
21 Therefore the output is 7.
22 Follow up:
23 What if negative numbers are allowed in the given array?
24 How does it change the problem?
25 What limitation we need to add to the question to allow negative
26 numbers?

```

Target as Climbing Stairs. Analysis: The DFS+MEMO solution is given in Section 13.1.2. 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):

```



Figure 21.5: 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 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     #initialize 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] #initialize 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]
```

21.4.2 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 21.6: Caption

```

6 How many distinct numbers can you dial in this manner?
7
8 Since the answer may be large , output the answer modulo  $10^9 +$ 
7 .
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
10     three
11
12     bfs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] # all starting points
13     step = 1
14     while bfs:
15         new = []
16         for i in bfs:
17             new += moves[i]
18         step += 1
19         bfs = new
20         if step == N:
21             return len(bfs)%(10**9+7)

```

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
7     three
8
9     bfs = [1]*10
10    step = 1
11
12    while bfs:
13        size = 0
14        new = [0]*10
15        for idx, count in enumerate(bfs):
16            for m in moves[idx]:
17                new[m] += count
18                new[m] %= MOD
19        step += 1
20        bfs = new
21        if step == N:
22            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 Input: 3, 2, 0, 0
- 7 Output: 0.0625
- 8 Explanation: There are two moves (to (1,2) , (2,1)) that will keep the knight on the board.
- 9 From each of those positions , there are also two moves that will keep the knight on the board.
- 10 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

```

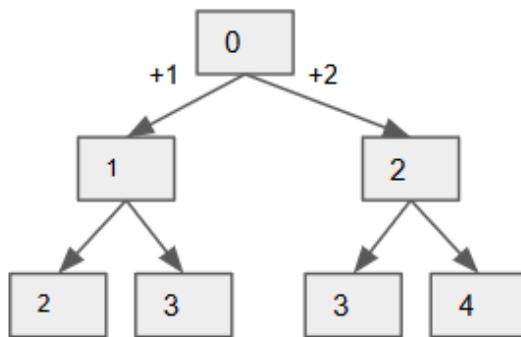


Figure 21.7: Tree Structure for One dimensional coordinate

BFS. Fig 21.7 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.

21.4.3 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 )

```

21.5 Double Sequence: Pattern Matching DP

	u_B	A	B	D
u_B	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 21.8: 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. 21.8. 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 (21.4.1).*

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. 21.8, 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 to
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)

21.5.1 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. 21.8. 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. 21.2. To initialize, we need to initialize the first row and the first column, which is $f[i][0] = 0$, $f[0][j] = 0$.

$$f[i][j] = \begin{cases} 1 + f[i-1][j-1], & a[i-1] == b[j-1]; \\ \max(f[i-1][j], f[i][j-1]) & \text{otherwise} \end{cases} \quad (21.2)$$

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]
12 S1 = "ABCD"
13 S2 = "ABD"
14 LCSLen(S1, S2)
15 # output
16 # [[0, 0, 0, 0], [0, 1, 1, 1], [0, 1, 2, 2], [0, 1, 2, 2], [0,
17 # 1, 2, 3]]
# 3

```

21.5.2 Other Problems

There are more pattern matching related dynamic programming, we give them in this section.

21.16 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 21.3. Thus, it can be solved by dynamic programming.

$$f[i][j] = \begin{cases} \min(f[i][j-1], f[i-1][j], f[i-1][j-1]) + 1, & S1[i-1]! = S1[j-1]; \\ f[i-1][j-1] & \text{otherwise} \end{cases} \quad (21.3)$$

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     for i in range(1, rows+1):
18         for j in range(1, cols+1):
19             if word1[i-1]==word2[j-1]:
20                 dp[i][j]=dp[i-1][j-1]
21             else:
22                 dp[i][j]=min(dp[i-1][j]+1, dp[i][j-1]+1, dp
23 [i-1][j-1]+1) # add, delete, replace
24     return dp[rows][cols]
```

21.17 115. 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 with-

out 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
 $\wedge\wedge\wedge \wedge\wedge$ 
rabbbit
 $\wedge\wedge \wedge\wedge\wedge$ 
rabbbit
 $\wedge\wedge\wedge \wedge\wedge\wedge$ 
```

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
 $\wedge\wedge \wedge$ 
babgbag
 $\wedge\wedge \wedge$ 
babgbag
 $\wedge \wedge\wedge$ 
babgbag
 $\wedge \wedge\wedge$ 
babgbag
 $\wedge\wedge\wedge$ 
```

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
a	1	0	0
b	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
32                        subsequence before this char in S is the same as t
33    return dp[-1][-1]
```

21.18 **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
13                 all '*'
14                 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
25                     all till the end
26                         #print(i)
27                         if helper(i , pi+1):
28                             return True
29                         return False
30             if p[pi] != s[si]:
31                 return False
32             return helper(si+1, pi+1)
33     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     or left           dp[ r ][ c ] = dp[ r -1][ c ] or dp[ r ][ c -1] # above
18
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]

```

21.5.3 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 i th element in the first string with the j th element in the second string;
3. initialize: $f[i][0]$ for the first column and $f[0][j]$ for the first row;
4. answer: $f[n][m]$

21.6 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 21.6.1): each item is only allowed to use 0 or 1 time.
2. Unbounded Knapsack(Section 21.6.2): each item is allowed to use unlimited times.
3. Bounded Knapsack(Section 21.6.3): 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 21.6.1, 21.6.2, and 21.6.3) 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 13.1.2 for the DFS based solution and reasoning.

LeetCode problems:

1. 322. Coin Change (***) unbounded, fixed amount.

21.6.1 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     ans = [0]
9     n = len(c)
10    dfs(0, 0, 0, ans)
11    return ans[0]
12
13 c = [1, 1, 2, 2]
14 v = [1, 2, 4, 5]
15 C = 4
16 print(knapsack01DFS(c, v, C))
17 # output
18 # 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]

```

21.6.2 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, ..., 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 $c[i]$, $v[i]$; $2^k c[i]$, $2^k v[i]$, 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]

```

21.6.3 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))
# 10

```

Reduce to Unbounded Knapsack. If $n[i] \geq C/c[i], \forall i$, then the Bounded Knapsack can be reduced to Unbounded Knapsack.

21.6.4 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: $f[i][0] = True; f[0][1, ..., size] = False$, 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 can't fill backpack with size in range (1, size).

4. Answer: $dp[n-1][C-1]$.

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 $\text{float}(" -\inf ")$ 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

```

21.6.5 LeetCode Problems

21.19 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 $\text{float}("inf")$ 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 sacpbac 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

```

21.20 **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 $dp[i-1][j-\text{nums}[i]]$. 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]
    return dfs(0, s//2)

```

21.7 Exercise

21.7.1 Single Sequence

Unique Binary Search Tree

Interleaving String

Race Car

21.7.2 Coordinate

746. Min Cost Climbing Stair (Easy)

```

1 On a staircase, the i-th step has some non-negative cost cost[i]
2   ] assigned (0 indexed).
3 Once you pay the cost, you can either climb one or two steps.
4   You need to find minimum cost to reach the top of the floor,
5   and you can either start from the step with index 0, or the
6   step with index 1.

```

```

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].

```

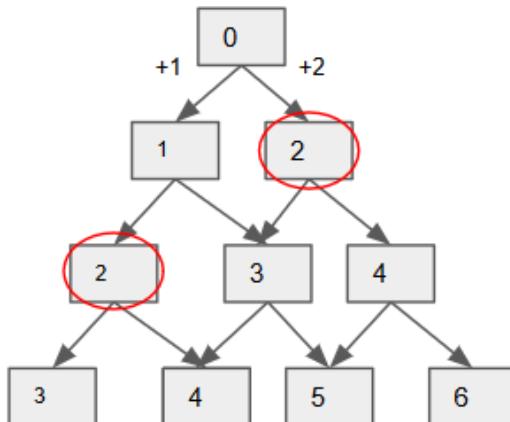


Figure 21.9: Caption

Analysis: As Fig 21.9 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]

```

```

1 There is an m by n grid with a ball. Given the start coordinate
2   (i, j) of the ball, you can move the ball to adjacent cell or
3   cross the grid boundary in four directions (up, down, left,
4   right). However, you can at most move N times. Find out the
5   number of paths to move the ball out of grid boundary. The
6   answer may be very large, return it after mod  $10^9 + 7$ .
7
8 Example 1:
9 Input: m = 2, n = 2, N = 2, i = 0, j = 0
10 Output: 6
11 Explanation:
12
13 Example 2:
14 Input: m = 1, n = 3, N = 3, i = 0, j = 1
15 Output: 12
16 Explanation:
17
18 Note:
19 Once you move the ball out of boundary, you cannot move it
20 back.
21 The length and height of the grid is in range [1,50].
22 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
22
23        dp = new_dp
24
25    return ans

```

63. Unique Paths II

```
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 Now consider if some obstacles are added to the grids. How many
7 unique paths would there be?
8 An obstacle and empty space is marked as 1 and 0 respectively in
9 the grid.
10 Note: m and n will be at most 100.
11 Example 1:
12 Input:
13 [
14     [0,0,0],
15     [0,1,0],
16     [0,0,0]
17 ]
18 Output: 2
19 Explanation:
20 There is one obstacle in the middle of the 3x3 grid above.
21 There are two ways to reach the bottom-right corner:
22 1. Right -> Right -> Down -> Down
23 2. Down -> Down -> Right -> Right

```

Coordinate.

```

1 def uniquePathsWithObstacles(self, obstacleGrid):
2     """
3         :type obstacleGrid: List[List[int]]
4         :rtype: int
5     """
6     if not obstacleGrid or obstacleGrid[0][0] == 1:
7         return 0
8     m, n = len(obstacleGrid), len(obstacleGrid[0])
9     dp = [[0 for c in range(n)] for r in range(m)]
10    dp[0][0] = 1 if obstacleGrid[0][0] == 0 else 0 # starting
11    point
12
13    # init col
14    for r in range(1, m):
15        dp[r][0] = dp[r-1][0] if obstacleGrid[r][0] == 0 else 0
16
17    for c in range(1, n):
18        dp[0][c] = dp[0][c-1] if obstacleGrid[0][c] == 0 else 0
19
20    for r in range(1, m):
21        for c in range(1, n):
22            dp[r][c] = dp[r-1][c] + dp[r][c-1] if obstacleGrid[r][c] == 0 else 0
23
24    print(dp)

```

```
23     return dp[-1][-1]
```

21.7.3 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 Example 1:
4
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 <= 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]
```


22

String pattern Matching Special

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 mismathces 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 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

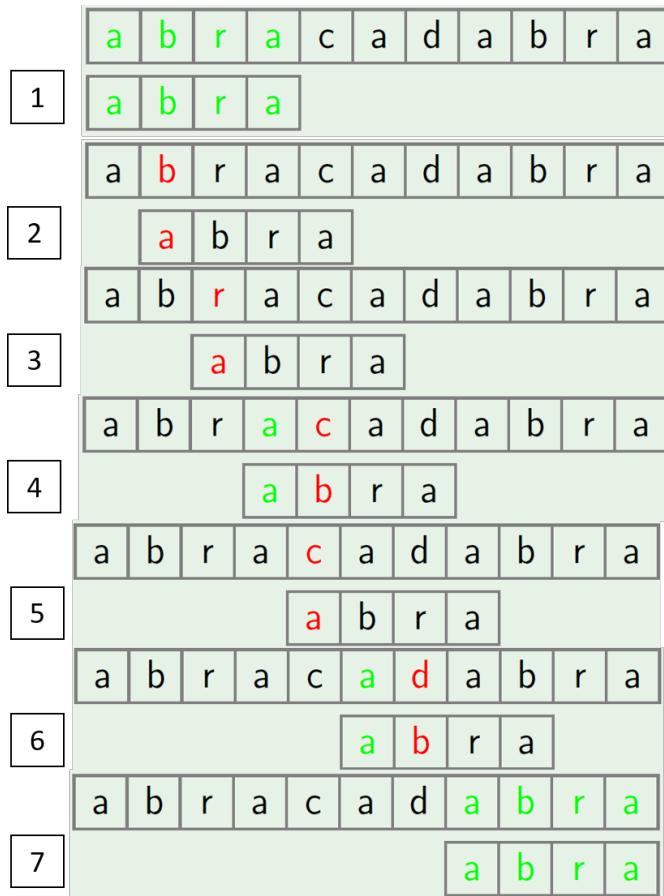


Figure 22.1: The process of the brute force exact pattern matching

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.

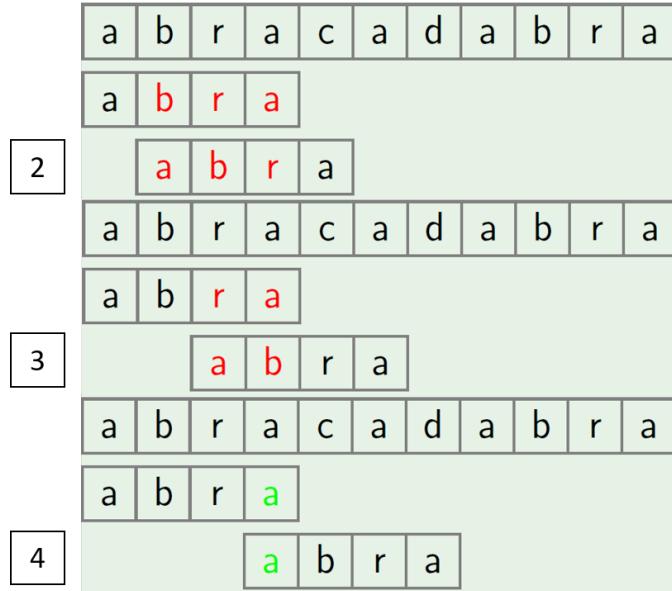


Figure 22.2: The Skipping Rule

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 'arba'
'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:

$$l[i] = \max_{k=0,\dots,i} \{k : P[0\dots k-1] = P[i-(k-1)\dots i]\} \quad (22.1)$$

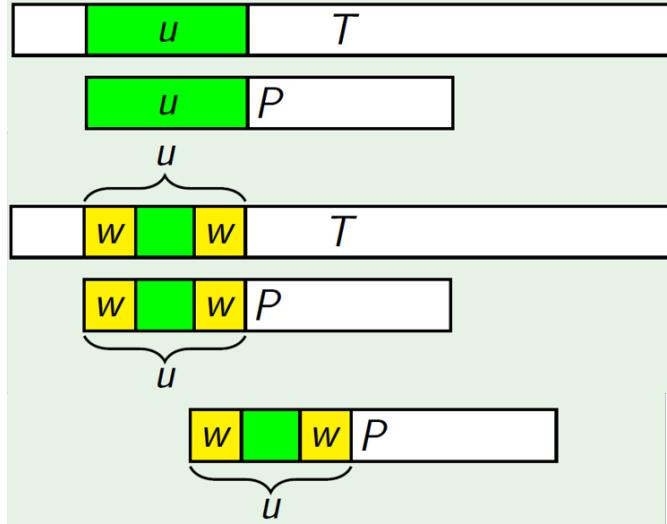


Figure 22.3: The Sliding Rule

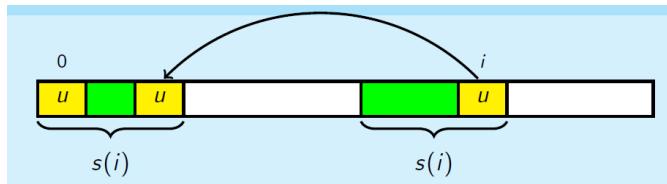


Figure 22.4: Proof of Lemma

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 $P[0\dots l(i)-1]$.

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 $P[0\dots i]$ (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 $s[0\dots \pi(i)]$ with $s[i - (\pi(i) - 1)\dots i]$, 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             j = pi[j-1]
9         # check the character
10        if s[i] == s[j]:
11            pi[i] = j + 1
12
13 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, ts):
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[pos] = 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 $z[i] = k, i \in [1, n - 1]$, which means the length of the longest common prefix between string s and the suffix of string starting at position i as $s[i \dots n - 1]$.

```
"aaaaa" - [0,4,3,2,1]
"aaabaab" - [0,2,1,0,2,1,0]
"abacaba" - [0,0,1,0,3,0,1]
```

To implement the z-function in $O(n^2)$ is trivial.

```

1 def naiveZF(s):
2     '''implement z function in O(n^2)'''
3     n = len(s)
4     z = [0] * n
5     for i in range(1, n): # starting point
6         k = i # starting index of the suffix s[i:]
7         while i + z[i] < n and s[i+z[i]] == s[z[i]]:
8             z[i] += 1
9     return z

```

Here, we show how we can implement it in $O(n)$. As shown in Fig. 22.5, at position i , l, r is the closest precede $z[k] = 0, k < i$. We know $s[l \dots r] = s[0, r - l]$ which is denoted as two $z[i]$ in the figure. Thus we know the μ

are marked with green color is equal too, to compare the second μ , which is $s[i \dots r]$ with string s is equivalent to compare the first μ , which is $s[i-l \dots r-l]$. And, the comparison between the first μ and the string is already computed and stored as $z[i-l]$. We name this as **Z-function property**. Therefore, we can set the initial value for $z[i] = z[i-l]$, and start the naive comparison from $s[z[i]]$ with $s[i+z[i]]$

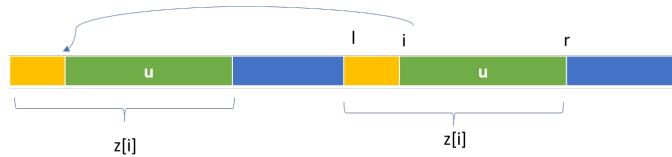


Figure 22.5: Z function property

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): # starting point
6         if i <= r:
7             z[i] = min(r-i+1, z[i - 1])
8             while i + z[i] < n and s[i+z[i]] == s[z[i]]:
9                 z[i] += 1
10            if i + z[i] - 1 > r:
11                l = i
12                r = i + z[i] - 1
13
14    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 $s[0...0]$, $s[0...1]$, ..., $s[1...i]$, ..., $s[0...n-1]$. 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 $s[0...i]$ and reverse it $i...0$. 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 $s[i...i-\max-1] = s[i-j...i-j+\max]$, which is to say $s[i-\max-1...i] = s[i-j-\max...i-j]$. 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' < 'abab', 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$', 'baa$', 'baba$']
[6, 5, 4, 2, 0, 3, 1]
```

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:babaaa\$	a\$ababa	a\$

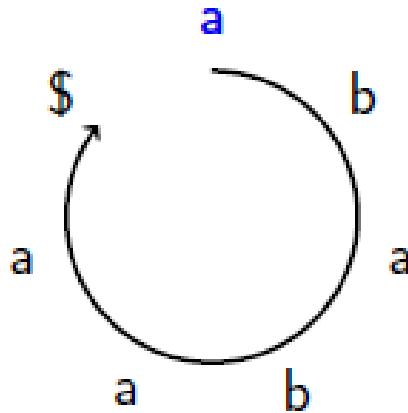


Figure 22.6: Cyclic Shifts

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@

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>ab</u>	<u>abab</u>
babaa\$a	b	<u>ba</u>	baba
abaa\$ab	a	<u>ab</u>	abaa
baa\$aba	b	<u>ba</u>	baa\$
aa\$abab	a	aa	aa\$
a\$ababa	a	a\$	a\$ba
\$ababaa	\$	<u>\$a</u>	\$aba

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 $O(n \log)$ ($T(n) = T(n/2)$) 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.

```

1 if s[order[i]] != s[order[i-1]]:
2     # use order as index to put the result back
3     class[order[i]] = class[order[i-1]] + 1
4 else:
5     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 getClass(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

```

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) %
n] != cls[(prev_order+L) % n]:

```

```

10     new_cls[cur_order] = new_cls[prev_order] + 1
11 else:
12     new_cls[cur_order] = new_cls[prev_order]
13 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 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

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 15.3. 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

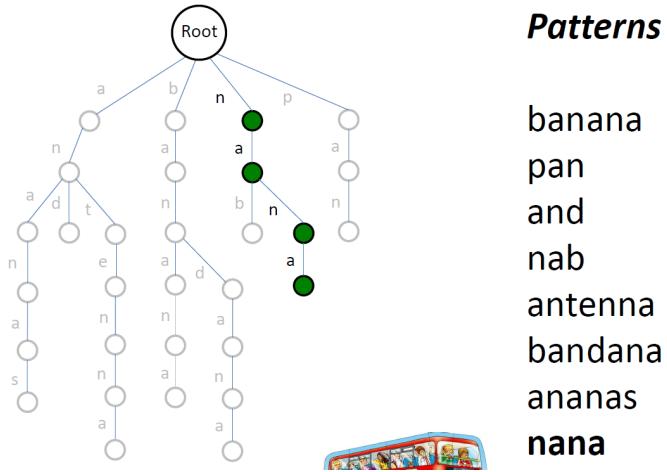


Figure 22.7: Building a Trie from Patterns

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.

Part VIII

Combinatorial Problems

Sorting and Selection Algorithms

In computer science, a **sorting algorithm** is designed to rearrange items of a given array of items in a certain order. 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 order:

```
Array = [9, 10, 2, 8, 9, 3, 7]  
sorted = [2, 3, 7, 8, 9, 9, 10]
```

Selection algorithm is an algorithm for finding the k -th smallest number in a given array; such a number is called the k -th *order statistic*. 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 algorithms from a corresponding sorting algorithm. Due to such relation, this chapter is mainly about introducing sorting algorithms and occasionally by the side of a proper sorting algorithm, we introduce its corresponding selection algorithm mutant.

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 that we will introduce in the

next book need the data to be in sorted order to carry on the next step.

How to Learn Sorting Algorithms? Before we start our journey to learn each individual existing sorting algorithm, it is worthy the time to discuss some key terminologies and techniques that distinguish different kind. Therefore, along the learning process, we know what questions to answer and trying to look for answer. Knowing the behavior and performance of each kind helps us making better decision when trying to design best solutions for real problems.

- **In-place Sorting:** In-place sorting algorithm only uses a constant number of extra spaces to assist its implementation.
- **Stable Sorting:** Stable sorting algorithm maintain the relative order of items with equal keys.
- **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 include this part into this chapter too.

Organization We organize the content mainly based on the worst case time complexity. Section 23.1 - 23.4 focuses on numerical sorting and selection algorithms. In addition, Section 23.5 completes the picture and show which sorting algorithms can be adapted to do lexicographical order sorting based on its distinct characters that the range of keys limited by $|\Sigma|$, the number of possible keys in the definition. Further, in Section 23.6, we introduce the built-in sort for list and function that applies on array data structures. Know the properties and how to customize the comparison functions. To recap:

- $O(n^2)$ Sorting (Section 23.1): Bubble Sort, Insertion Sort, Selection Sort;
- $O(n \log n)$ (Section 23.2) Sorting: merge sort, quick sort, and Quick Select;

- $O(n + k)$ Sorting (Section 23.3): Counting Sort, where k is the range of the very first and last key.
- $O(n + k)$ Sorting (Section 23.4): Bucket Sort and Radix Sort.
- Lexicographical Order (Section 23.5): In this section, the algorithms that do lexicographical order is
- Python Built-in Sort (Section 23.6):

23.1 $O(n^2)$ Sorting

As the most naive and intuitive group of comparison-based sorting methods, this group takes $O(n^2)$ time and are usually consist of two nested for loops. In this section, we learn three different ones “quickly” due to their simplicity: bubble sort, insertion sort and selection sort.

23.1.1 Insertion Sort

Insertion sort is one of the most intuitive sorting algorithms for humans. For humans, with an array of n items to process, each time we take one item “out” to another processed list by inserting it at the right position. At first, the processed list is empty, then with one item, and in both case, it is naturally sorted. Through insert the new item at the right position, the processed list is maintained to be sorted all the time. After the very last item is inserted into the sorted sublist, the sorted sublist will be the final sorted array of the given input. The whole process on given array $a = [9, 10, 2, 8, 9, 3]$ looks like Fig. 23.1.

In-place Insertion We can either use extra space or choose to do it in-place. Here we discuss how we can do the in-place insertion. A key step in insertion sort is to insert a target item a_i into the sorted sublist. We iterate through the sorted sublist $a[0...i - 1]$. There are two different ways for iteration: forward and backward. We use pointer j for the sublist.

- Forward: j will iterate in range $[0, i-1]$. We compare $a[j]$ with $a[i]$, we stop at the first place that $a[j] > a[i]$ (to keep it stable), and all elements $a[j...i-1]$ will be shifted to backward, and $a[i]$ will be placed at index j . Here we need completely of i comparison and swaps.
- Backward: j iterates in range $[i-1, 0]$. We compare $a[j]$ with $a[i]$, we stop at the first place that $a[j] <= a[i]$ (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.

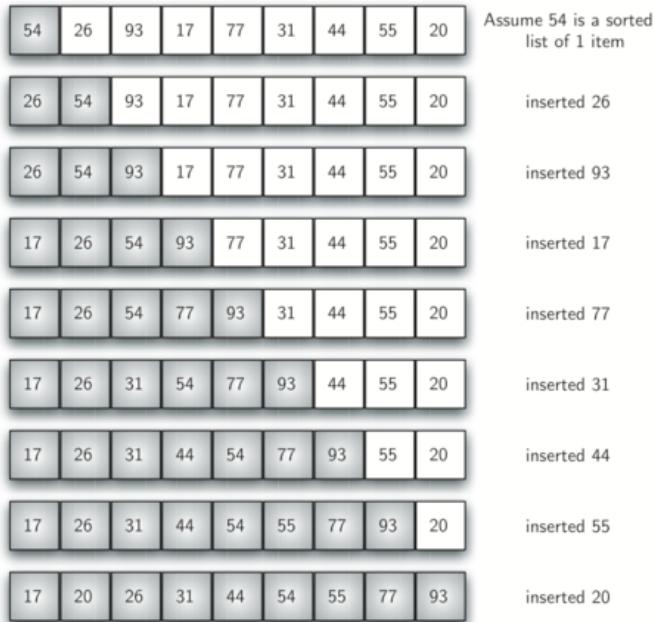


Figure 23.1: The whole process for insertion sort

Implementation Using extra space to save the sorted sublist is the easy start. We use the first for loop to indicate the item to be inserted, and the second for loop to find the right position and do insertion in the sorted sublist.

```

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): # items to be inserted into the sorted
7         j = 0
8         while j < len(sl):
9             if a[i] > sl[j]:
10                 j += 1
11             else:
12                 sl.insert(j, a[i])
13                 break
14             if j == len(sl): # not inserted yet
15                 sl.insert(j, a[i])
16     return sl

```

We can also do it in-place by shifting as has shown above. Implementing with backward:

```

1 def insertionSortInPlace(a):
2     if not a or len(a) == 1:
3         return a

```

```

4     n = len(a)
5     for i in range(1, n): # items to be inserted into the sorted
6         t = a[i]
7         j = i - 1
8         while j >= 0 and t < a[j]: # keep comparing if target is
9             a[j+1] = a[j] # shift current item backward
10            j -= 1
11        a[j+1] = t # a[j] <= t , insert t at the location j+1
12    return a

```

23.1.2 Bubble Sort and Selection Sort

Bubble Sort

Bubble sort less intuitive as of the insertion sort to human, it compares each pair of adjacent items in an array and swaps them if they are out of order.

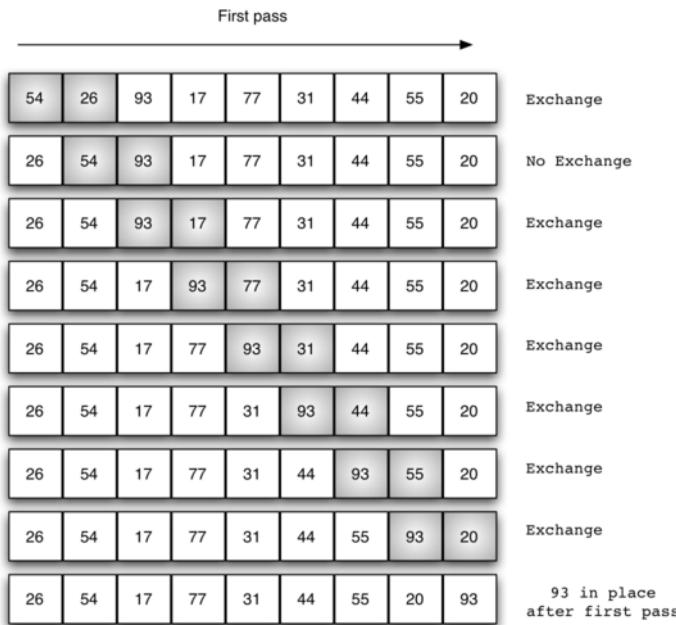


Figure 23.2: One pass for bubble sort

One Pass Given an array of size n , in a single pass, there are $n - 1$ pairs waiting for comparison and with potential $n - 1$ times of swap operations. For example, when the array is: [9, 10, 2, 8, 9, 3, 7], to sort them in ascending order. When comparing a pair (A_i, A_{i+1}) , if $A_i > A_{i+1}$, we swap these two items. One pass for the bubble sort is shown in Fig. 23.2, and we can clearly see after one pass, the largest item will be in place. This is what “bubble”

means in the name that each pass, the largest item in the valid window bubble up to the end of the array.

Next Pass Therefore, in the next pass, the last item will no longer needed to be compared. For pass i , it places the current i -th largest items in position in range of $[n - i - 1, n)$. the last i items will be sorted. We say, in the first pass, where $i = 0$, the valid window is $[0, n)$, and to be generalize the valid window for i -th pass is $[0, n - i)$.

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 pass, which is total $n-1$; the second for loop to index the starting index of each pair in the valid window:

```

1 def bubbleSort(a):
2     if not a or len(a) == 1:
3         return a
4     n = len(a)
5     for i in range(n - 1): #n-1 passes ,
6         for j in range(n - i - 1): #each pass will have valid
7             window [0, n-i] , and j is the starting index of each pair
8                 if a[j] > a[j + 1]:
9                     a[j], a[j + 1] = a[j + 1], a[j] #swap
10    return a

```

In the code, we use $a[j] > a[j + 1]$, because when there 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**. As we see in bubble sort, there is no extra space needed to sort, this makes it **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 $T = \sum_{i=0}^{n-1} (n - i - 1) = n - 1 + (n - 2) + \dots + 2 + 1 = n(n - 1)/2 = O(n^2)$. 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 pass. This can achieve better solution as good ad $O(n)$ if the input is nearly sorted.

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. Selection sort makes a slight optimization through selecting the largest item in the current valid window and swap it directly with the item at its corresponding right position. This avoids the constantly swapping operations in the bubble sort. Each pass we

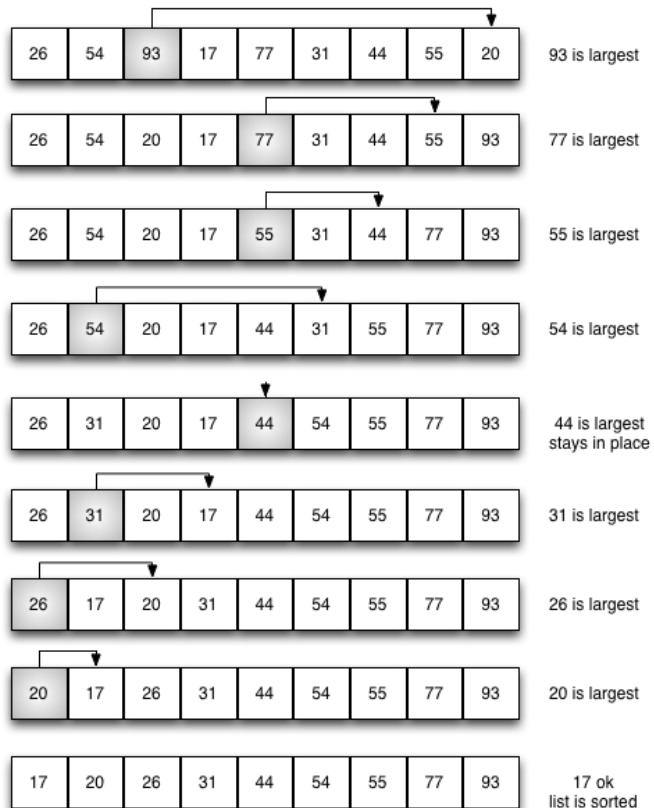


Figure 23.3: The whole process for Selection sort

find the largest element and switch it with the last element. Then the next pass has one less element to loop through. The process is shown in Fig 23.3.

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 valid window at the inner for loop. We use a variables ti and li to record 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 # the position to fill in the largest item of
5         valid window [0, n-i]
6         li = 0
7         for j in range(n - i):
8             if a[j] > a[li]:
9                 li = j
10            # swap li and ti
11            a[ti], a[li] = a[li], a[ti]
12
13    return a

```

Like bubble sort, selection sort is **in-place**. Given an array [9, 10, 2, 8, 9, 3, 9], there exists equal keys 9. At the first pass, the very last 9 is swapped to the position 1 with key 10, and it becomes the second among its equals. Unlike bubble sort, this mechanism makes selection sort **unstable**.

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.

23.2 $O(n \log n)$ Sorting

We have learned a few comparison-based sorting algorithms and they all have upper bound of n^2 in the number of comparisons must be executed. Think about the lower bound of complexity for comparison-based sorting, and ask such questions: 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 by comparing a pair of items each time. So, how many times of comparison do we need? Let's try the case when $n = 3$, and all possible permutations 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 $a_1 < a_2$, it will narrow down to only three choices (1, 2, 3), (1, 3, 2), (3, 1, 2). This is a decision-tree model, each branch represents one decision made on the comparison result, and each time it narrows down the choice to half. The height h of the binary decision-tree. And because it will have at most 2^h leaves which represent one possible in the permutations set $n!$. Therefore we get the inequation:

$$2^h \geq n!, h \geq \log(n!)h = \Omega(n \log n) \quad (23.1)$$

In the remaining content of this section, we will introduce the most general and commonly used sorting algorithms that has $O(n \log n)$ that matches our approved lower bound. Merge Sort and Quick Sort both utilize the Divide-and-conquer method. Heap Sort on the other hand uses the max/min heap data structures we have learned before to get the same upper bound performance.

23.2.1 Merge Sort

We have already used Merge Sort as the basic example of illustrating the Divide-and-Conquer algorithm design methodology. As we know there are two main steps: “divide” and “merge” in merge sort and each happens at a different stage of the recursion function call.

Divide In the divide stage, the original problem $a[s...e]$, s, e is the start and end index of the subarray. The divide process divides its parent problem into two halves: $a[s...m]$, $a[m + 1, e]$. This recursive call keeps moving downward till the size of the subproblem becomes one where $s = e$, which is the base case and the sublist of one item is naturally sorted. The process of divide is shown in Fig. 23.4.

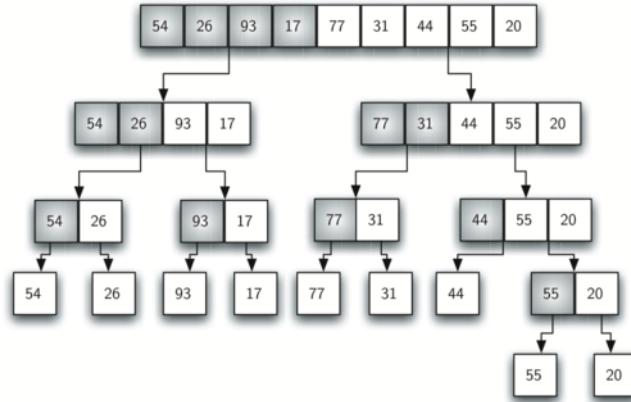


Figure 23.4: Merge Sort: divide process

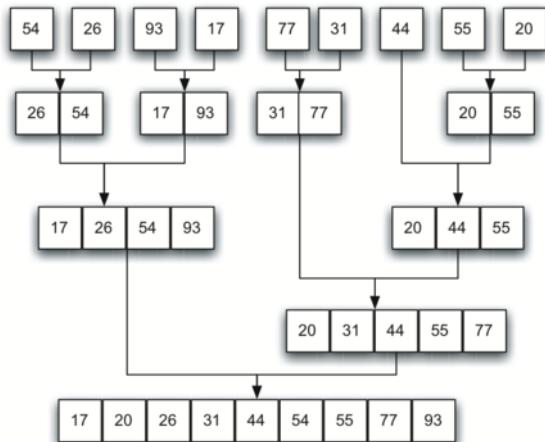


Figure 23.5: Merge Sort: merge process

Merge After the return from our subproblems with its solution, we merge the sorted left l and right sorted sublist r . The comparison happens in the merge stage. We can put two pointers at the start of l and r , and each time choose the smaller item between the two until any pointer reaches to the end. The subprocess is shown in Fig. 23.5 and its implementation is as

follows:

```

1 def merge(l, r):
2     '''combine the left and right sorted list'''
3     ans = []
4     i = j = 0 # two pointers each points at l and r
5     n, m = len(l), len(r)
6
7     # first while loop to merge
8     while i < n and j < m:
9         if l[i] <= r[j]:
10             ans.append(l[i])
11             i += 1
12         else:
13             ans.append(r[j])
14             j += 1
15
16     # now one list of l and r might have items left
17     ans += l[i:]
18     ans += r[j:]
19     return ans

```

In the code, we use $l[i] \leq r[j]$ instead of $l[i] < r[j]$ is because when there are items of equal keys, we should put the ones in the left first in the merged list. Therefore, our merge sort is **stable**.

Implementation The whole implementation is straightforward.

```

1 def mergeSort(a, s, e):
2     # base case , can not be divided further
3     if s == e:
4         return [a[s]]
5     # divide into two halves from the middle point
6     m = (s + e) // 2
7
8     # conquer
9     l = mergeSort(a, s, m)
10    r = mergeSort(a, m+1, e)
11
12    # combine
13    return merge(l, r)

```

Check out the sample code to how to prove a sorting is stable or not using tuple. In the merge process, we use a temporary space to save the merged result and return it to its last recursive call, thus merge sort is **not in-place**, and it takes extra space which is $O(n)$ to implement.

Complexity Analysis Because each divide we need to take $O(n)$ time to merge the two lists back to a list. The complexity function can be deducted

as follows:

$$\begin{aligned}
 T(n) &= 2T(n/2) + O(n) \\
 &= 2 * 2T(n/4) + O(n) + O(n) \\
 &= O(n \log n)
 \end{aligned} \tag{23.2}$$

23.2.2 HeapSort

To sort the given array in increasing order, we can use max-heap and firstly heapify the given array. Since the maximum item is stored at the root, we can put swap root with the last item in the list, and then we can view the heap as a smaller size of $n - 1$. We observe all the subtree of the root remain max-heaps other than the new root item that might violate the max-heap property. We restore the max-heap property through sinking, and put the root item at position $n - 2$, and repeat the process for $n-1$ times.

Implementation Continuing use our exemplary Heap class in Chapter 10, we implement heapsort member function:

```

1 def heapsort(self, a):
2     self.heapify_sink(a)
3     n = len(a)
4     for i in range(n, 1, -1): # position to put the root node
5         self.heap[i], self.heap[1] = self.heap[1], self.heap[i]
#swap root with i
6         self.size -= 1
7         self._sink(1) # sink down the new root
8         print(self.heap)

```

We can also implement with heapq built-in data structure using the heapq and through the heappush() and heappop() function. :

```

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 first way of heapsort, the heapify_sink takes $O(n)$, and the later process takes $O(\log n + \log n - 1 + \dots + 0) = \log(n!)$ which has an upper bound of $O(n \log n)$. Check out more on <https://www.programiz.com/dsa/heap-sort>.

23.2.3 Quick Sort and Quick Select

Like merge sort, quick sort uses divide and conquer method and mainly implemented with recursion. Unlike merge sort, the key of the sorting called **partition** happens before the “divide” process in the recursive call smaller problems stage of the recursion.

Partition and Pivot Unlike merge sort which simply divide the list into half and half, quick sort chooses a **pivot**. Given a subarray of $A[s...e]$, the pivot can either locate at s/e or randomly chosen each time. Then quick sort partition $A[s...e]$ into three parts: $A[s...p - 1], A[p], A[p + 1...e]$, where $A[p]$ is the pivot, and for $A[i] \leq A[p], i \in [s, p - 1]$, and $A[i] > A[p], i \in [p + 1, e]$. And this partition process is the key step in the whole quick sort algorithm. There are normally two different partition methods: Lomuto's partition and Hoare's Partition which we will detail on later.

Conquer In the partition stage, one item will be in place, which is $A[p]$. Therefore, we only need to handle to smaller problems: sorting $A[s...p]$ and $A[p + 1...e]$ by recursively call the quicksort function.

Implementation Without considering the partition process, we can write down the main steps of quick sort as:

```

1 def quickSort(a, s, e):
2     # base case , can not be divided further
3     if s >= e:
4         return
5     p = partition(a, s, e)
6
7     # conquer smaller problem
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. Here, we assume using $A[e]$ as pivot. We would use two pointer technique i, j to maintain three regions in subarray $A[s...e - 1]$ as of: (1) region with items smaller than or equal to $A[e]$; (2) region with item larger than $A[e]$; (3) unrestricted region.

Lomuto's Partition

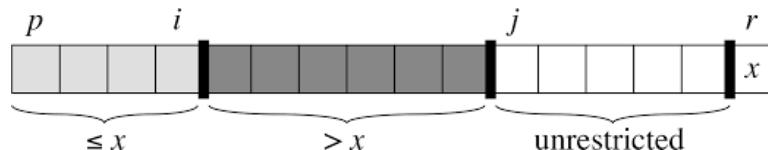


Figure 23.6: Lomuto's Partition: Three regions in the process, $A[p...i]$, $A[i+1...j]$, $A[j+1...e-1]$. At position $j+1$, if $A[j+1] \leq x$, then $A[j+1]$ and $A[i+1]$ is swapped and i and j move forward one location.

For lomuto's partition, we first place i at $s - 1$, and j at s . In this process, we move i and j to maintain the area as shown in Fig. 23.6, where

$A[s...i] \leq A[e]$ and $A[i+1...j] > A[e]$, and the unrestricted area $A[j+1...e-1]$. i points to the end of area one, and j points to the start of area three. At the start, area one and one are both empty. And the unrestricted area is $A[s, e-1]$. We scan item in the unstricted area using pointer j . If the current item $A[j]$ belongs to area two, that is to say $A[j] > A[e]$, we just increment pointer j ; otherwise, this item should goes to area one, we accomplish this by swapping this item with the first item in the current area two, at $i + 1$, therefore, the area increments by one. After the for loop, we need to put our pivot at the first place of area two by swapping. And now, the whole subarray is successfully partitioned into three regions as we needed, and return $i+1$ as the partition index. The implementation of as follows:

```

1 def partition(a, s, e):
2     p = a[e]
3     i = s - 1
4     for j in range(s, e): #a[s, e-1]
5         if a[j] <= p:
6             i += 1
7             a[i], a[j] = a[j], a[i] # swap a[i] and a[j]
8     # place p at position i+1 through swapping with a[i+1]
9     a[i+1], a[e] = a[e], a[i+1]
10    return i+1

```

Complexity Analysis The worst case of partition appears when the input array is already sorted or is reversed from the sorted array. In this case, it will partition one subproblem with size n into $n - 1$ and another just empty. The recurrence function is $T(n) = T(n - 1) + O(n)$, and it will have a time complexity of $O(n^2)$. And the best case appears when a subprocess is divided into half and half as the merge sort, and the time complexity will be $O(n \log n)$. Randomly picking the pivot from $A[s...e]$ and swap it with $A[e]$ can help us achieve an average and stable performance.

Properties 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 two, it can be swapped backward. (2) as the pivot, it will be swapped with the first item in the region two too. Therefore, it is hard to guarantee the stability among equal keys. We can try experiment with $a = [(5, 1), (7, 1), (3, 1), (2, 1), (5, 2), (6, 1), (7, 2), (8, 1), (9, 1), (5, 3), (5, 4)]$, and use the first element in the tuple as key.

Quick Select

Quickselect is a selection algorithm to find the k -th smallest element in an unordered list. It is related to the quick sort sorting algorithm.

The algorithm is similar to QuickSort. The difference is, instead of recurring for both sides (after finding pivot), it recurs only for the part that contains the k-th smallest element. The logic is simple, if index of partitioned element is more than k, then we recur for left part. If index is same as k, we have found the k-th smallest element and we return. If index is less than k, then we recur for right part. This reduces the expected complexity from $O(n \log n)$ to $O(n)$, with a worst case of $O(n^2)$.

```

1 def quickSelectHelper(alist, first, last, k):
2     if first < last:
3         splitpoint = lomo_partition(alist, first, last)
4
5         if k == splitpoint:
6             return alist[k]
7         elif k < splitpoint: # find them on the left side
8             return quickSelectHelper(alist, first, splitpoint - 1,
9                                     k)
10        else: # find it on the right side
11            return quickSelectHelper(alist, splitpoint + 1, last, k)
12
13 def quickSelect(alist, k):
14     if k > len(alist):
15         return None
16     return quickSelectHelper(alist, 0, len(alist) - 1, k - 1)

```

Now call the function as:

```

1 alist = [54, 26, 93, 17, 77, 31, 44, 55, 20, 100]
2 print('After quick select of the 3rd smallest item: ', quickSelect(alist, 3))

```

The output is:

```
1 After quick select of the 3rd smallest item: 100
```

23.3 $O(n + k)$ Counting Sort

For a sequence of integers that are in the range of k , Counting sort is a linear time sorting algorithm that sort in $O(n + k)$ time. So counting sorting in its fitting occasion, it took liner time, but for the integers that are in the range of n^2 , then the time complexity would be $O(n^2)$, which means we better use some other sorting algorithms, either the $O(n \log n)$ or the radix sorting that is going to be introduced in the next section.

The counting sort is a sorting technique based on keys between a specific range. It works by counting the occurrence number of each distinct key and saves it into a count array. Then a prefix sum computation is applied on count array. Eventually, a loop over the given array and put the key to location pointed out by the count array.

Let us understand each step with the help of an example. For simplicity, consider the data in the range 0 to 9. The input data has duplicates and each is distinguished by the order in the parentheses.

Input data
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

Step 1: Count Occurrence We assign a count array $C_i, i \in [0, k - 1]$ which has the same size of the key range k , and loop over the input data to count each key's occurrence times in the input. In our example, we have the following result for the count array:

Index:	0	1	2	3	4	5	6	7	8	9
Count:	0	2	2	0	1	1	0	1	0	0

Step 2: Prefix Sum on Count Array. We apply the formula $C_i = C_i + C_{i-1}$, for $i \in [1, k - 1]$ to obtain the position range of each key in the input array. For key i , its sorted order in the input will be $(C_{i-1}, C_i]$. Observing the count array after this step showing in the below table, key 7 will be the order $(C_6, C_7]$, which is $(6, 7]$, that is only the 7-th position at index 6. For key 2, it is in the range of $(2, 4]$, that is 3-th and 4-th position.

Index:	0	1	2	3	4	5	6	7	8	9
Count:	0	2	4	4	5	6	6	7	7	7

Step 3: Put Keys to Positions Pointed by Count Array. Knowing the meaning of the current count array, we can put each key at position at $C[i]$ and then decrease $C[i]$ by one so that the next appearance of the key can have the right position. First, let us loop over the input keys from position 0 to $n - 1$. The corresponding output is shown in follows, and we can see the order of the same key is reversed, this makes the sorting unstable.

Index:	0	1	2	3	4	5	6
Key:	1(1)	4	1(2)	2(1)	7	5	2(2)
Sorted:	1(2)	1(1)	2(2)	2(1)	4	5	7

In order to correct this and be stable, we loop over each key in the input data in reverse order instead.

Implementation With the understanding of the algorithm, the implementation is trivial and we increase the adaptation by finding the range of the keys. The Python code is given here, we also encourage your to play the code around.

```

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
11    # get prefix sum
12    for i in range(1, k):
13        count[i] += count[i-1]
14
15    # put it back in the input
16    for i in range(n-1, -1, -1):
17        key = a[i] - minK
18        count[key] -= 1 # to get the index as position
19        order[count[key]] = a[i] # put the key back to the sorted
20        position
21    return order

```

Properties

23.4 $O(n)$ Sorting

There are sorting algorithms that are not based on comparisons, e.g. Bucket Sort and Radix Sort. These algorithms draw more than 1 bit of information from each step. Therefore, the information theoretic lower bound is not likewise a lower bound for the time complexity of these algorithms. In fact, Bucket Sort and Radix Sort have a time complexity of $O(n)$. However, these algorithms are not as general as comparison based algorithms since they rely on certain assumptions concerning the data to be sorted.

23.4.1 Bucket Sort

Bucket sort is a comparison sort algorithm that operates on elements by dividing them into different buckets and then sorting these buckets individually. Each bucket is sorted individually using a separate sorting algorithm or by applying the bucket sort algorithm recursively. Bucket sort is mainly useful when the input is uniformly distributed over a range.

Assume one has the following problem in front of them:

One has been given a large array of floating point integers lying uniformly between the lower and upper bound. This array now needs to be sorted. A simple way to solve this problem would be to use another sorting algorithm such as Merge sort, Heap Sort or Quick Sort. However, these algorithms guarantee a best case time complexity of $O(N \log N)$. However, using bucket

sort, the above task can be completed in (N) time. For example, consider the following problem.

Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

To sort the array in linear time? Counting sort can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers. The idea is to use bucket sort. Following is bucket algorithm.

```

1 bucketSort(arr [], n)
2 1) Create n empty buckets (Or lists).
3 2) Do following for every array element arr[i].
4     .....
5     a) Insert arr[i] into bucket[n*array[i]]
6 3) Sort individual buckets using insertion sort.
7 4) Concatenate all sorted buckets.

```

Python code:

23.4.2 Radix Sort

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort. The algorithm is demonstrated with the following exmaple:

```

1 Original , unsorted list :
2
3      170, 45, 75, 90, 802, 24, 2, 66
4
5 Sorting by least significant digit (1s place) gives :
6
7      170, 90, 802, 2, 24, 45, 75, 66
8
9 Sorting by next digit (10s place) gives :
10
11     802, 2, 24, 45, 66, 170, 75, 90
12
13 Sorting by most significant digit (100s place) gives :
14
15     2, 24, 45, 66, 75, 90, 170, 802

```

To implement the code with Python, we need to know how to get the each digit from the least significant to the most significant digit. We know if we divide an integer by 10, and the remainder would be the least significant digit. Then we divide this integer by 10, and get the remainder of this value of 10, then we would have the digit for the least significant digit. The Python code is shown as follows:

```

1 a = 178
2 while a>0:
3     digit = a%10

```

```
4     a /= 10
```

The result of this digit would be 8, 7, 1. The Python code:

```

1 # Python program for implementation of Radix Sort
2
3 # A function to do counting sort of arr[] according to
4 # the digit represented by exp.
5 def countingSort(arr, exp1):
6
7     n = len(arr)
8
9     # The output array elements that will have sorted arr
10    output = [0] * (n)
11
12    # initialize count array as 0
13    count = [0] * (10)
14
15    # count the digit at the same position, which is index%10
16    for i in range(0, n):
17        index = (arr[i]/exp1) #exp1 keeps multiply by 10 to get
18        rid of the dealt digit
19        count[ (index)%10 ] += 1
20
21    # Change count[i] so that count[i] now contains actual
22    # position of this digit in output array
23    for i in range(1,10):
24        count[i] += count[i-1]
25
26    # Build the output array
27    i = n-1
28    while i>=0:
29        index = (arr[i]/exp1)
30        output[ count[ (index)%10 ] - 1 ] = arr[i]
31        count[ (index)%10 ] -= 1
32        i -= 1
33
34    # Copying the output array to arr[],
35    # so that arr now contains sorted numbers
36    i = 0
37    for i in range(0,len(arr)):
38        arr[i] = output[i]
39
40 # Method to do Radix Sort
41 def radixSort(arr):
42
43     # Find the maximum number to know number of digits
44     max1 = max(arr)
45
46     # Do counting sort for every digit. Note that instead
47     # of passing digit number, exp is passed. exp is  $10^i$ 
48     # where i is current digit number
49     exp = 1
50     while max1/exp > 0:
51         countingSort(arr,exp)
```

```

51     exp *= 10
52
53 # Driver code to test above
54 arr = [ 170, 45, 75, 90, 802, 24, 2, 66]
55 radixSort(arr)
56
57 for i in range(len(arr)):
58     print(arr[i]),
59
60 # This code is contributed by Mohit Kumra

```

Here, we give a comprehensive summary of the time complexity for different sorting algorithms. The connection of BST/Trie to Quicksort/Radix

	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 23.7: The time complexity for common sorting algorithms

Sort.

Need further understanding. A binary search tree is a dynamic version of what happens during quicksort. The root represents an arbitrary (but hopefully not too far off from the median) pivot element from the collection. The left subtree is then everything less than the root, and the right subtree is everything greater than the root. The left and right collections are then again ordered in the same manner, i.e. the data structure is defined recursively.

A trie is a dynamic version of what happens during radix sort. You look at the first bit or digit of a number (or first letter of a string) to determine which subtree the value belongs in. You then repeat the procedure recursively using the next character or digit to determine which of the subtree's children it belongs in, and so on.

23.5 Lexicographical Order

For a list of strings, sorting them will make them in lexicographical order. The order is decided by a comparison function, which compare corresponding

characters of the two strings from left to right, and the first character where the two strings differ determines which string comes first. Or when string s is a prefix of string t, then s is smaller than t. 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' (differ at i = 0)
'abc' < 'abd' (differ at i = 2)
'ab' < 'abab' ('ab' is a prefix of 'abab')
```

Use Counting Sort Counting sort is one of the most efficient sorting algorithm for lexicographical ordered sorting.

```
1 def countSort(arr):
2
3     # The output character array that will have sorted arr
4     output = [0 for i in range(256)]
5
6     # Create a count array to store count of individual
7     # characters and initialize count array as 0
8     count = [0 for i in range(256)]
9
10    # For storing the resulting answer since the
11    # string is immutable
12    ans = [" " for _ in arr]
13
14    # Store count of each character
15    for i in arr:
16        count[ord(i)] += 1
17
18    # Change count[i] so that count[i] now contains actual
19    # position of this character in output array
20    for i in range(256):
21        count[i] += count[i-1]
22
23    # Build the output character array
24    for i in range(len(arr)):
25        output[count[ord(arr[i])]-1] = arr[i]
26        count[ord(arr[i])] -= 1
27
28    # Copy the output array to arr, so that arr now
29    # contains sorted characters
30    for i in range(len(arr)):
31        ans[i] = output[i]
32    return ans
33
34 # Driver program to test above function
35 arr = "geeksforgeeks"
36 ans = countSort(arr)
37 print "Sorted character array is %s" %("".join(ans))
```

23.6 Python Built-in Sort

There are two ways to use Python built-in sorting function: (1) the method function of list `lst.sort()` and (2) Python built-in function `sorted()`.

```
1 list . sort (key =... , reverse =...)
2 sorted (list , key =... , reverse =...)
```

23.6.1 Basic and Comparison

These two methods both using the same sorting method – *Timsort* 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.

The parameters of these two methods are exactly the same:

- Parameter *key*: the value of *key* will be any function that allows to sort on the basis of the value returned from this function.
- Parameter *reverse*: Boolean, If True, the sorted list is in Descending order. The default value is False.

The difference of these two methods are: `sort()` basically works with the *list* itself. It modifies the original list in place. The return value is *None*; `sorted()` works on *any iterable* that may include list, string, tuple, dict and so on. It returns another list and doesn’t modify the original input.

Let us first see some examples, using them and then focusing on how we can customize the comparison function through “key”.

Basic Examples

First, try sort a given list of integers/strings in-place:

```
1 lst = [4, 5, 8, 1, 2, 7]
2 lst . sort ()
3 print (lst)
4 # output
5 # [1, 2, 4, 5, 7, 8]
6 lst . sort (reverse=True)
7 # [8, 7, 5, 4, 2, 1]
```

Second, try sort a given tuple/dictionary of integers/strings using `sorted()`:

```

1 tup = (3, 6, 8, 2, 78, 1, 23, 45, 9)
2 sorted(tup)
3 #[1, 2, 3, 6, 8, 9, 23, 45, 78]

```

Note: For lists, 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.

23.6.2 Customize Comparison Through Key

Before we move to the next section, we need to answer two questions: (1) how does 'key' argument work? The purpose of key is to specify a function to be called on each list element **prior** to making comparisons. We have multiple choices to customize the key argument: (1) through lambda function, (2) through a pre-defined function, (3) through the

Comparison Through Function We can use either lambda function or a normal function to specify the key that we want to apply for the sorting. For example, given a list of tuples, as :

```

1 lst = [(1, 8, 2), (3, 2, 9), (1, 7, 10), (1, 7, 1), (11, 1, 5),
         (6, 3, 10), (32, 18, 9)]

```

We want to compare them only with the first element in each tuple.

```

1 def getKey(item):
2     return item[0]
3 sorted(lst, key = getKey)
4 #[((1, 8, 2), (1, 7, 10), (1, 7, 1), (3, 2, 9), (6, 3, 10), (11,
      1, 5), (32, 18, 9))]

```

Therefore, the comparison treats tuples such as (1, 8, 2), (1, 7, 10), (1, 7, 1) has the same value, and the sorted order following their original order in the lst. For simplicity, the same thing can be done with lambda function.

```

1 sorted(lst, key = lambda x: x[0])

```

This can be translated like this: for each element (x) in mylist, return index 0 of that element, then sort all of the elements of the original list 'mylist' by the sorted order of the list calculated by the lambda function.

Similarly, if we want to sort them with a lexicographical order: first by the first element, second by the second element but in reversed order, and third by the last element, we can return a tuple given a input x as (x[0], -x[1], x[2]) :

```

1 sorted_lst = sorted(lst, key = lambda x: (x[0], -x[1], x[2]))
2 # output
3 #[((1, 8, 2), (1, 7, 1), (1, 7, 10), (3, 2, 9), (6, 3, 10), (11,
      1, 5), (32, 18, 9))]

```

Same rule applies to objects with named attributes. For example:

```

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))

1 student_objects = [Student('john', 'A', 15), Student('jane', 'B',
2 , 12), Student('dave', 'B', 10)]
2 sorted(student_objects, key=lambda x: x.age)      # sort by age
3 [(dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

```

Operator Module Functions 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(), attrgetter(), and a methodcaller() function. Using those functions, the above examples become simpler and faster:

```

1 >>> from operator import itemgetter, attrgetter
2 >>> sorted(student_tuples, key=itemgetter(2))
3 [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
4 >>> sorted(student_objects, key=attrgetter('age'))
5 [(dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

```

The operator module functions allow multiple levels of sorting. For example, to sort by grade then by age:

```

1 >>> sorted(student_tuples, key=itemgetter(1,2))
2 [('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
3
4 >>> sorted(student_objects, key=attrgetter('grade', 'age'))
5 [('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

```

Customize Comparison Through Class's Method Function cmp specifies a custom comparison function of two arguments (list items) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument: cmp=lambda x,y: cmp(x.lower(), y.lower()). The default value is None.

First, we can define a class that has `__lt__` function, which is how the sorting algorithm key works.

```

1 class LargerNumKey(str):
2     def __lt__(x, y):
3         return x+y > y+x
4 class Solution:
5     def largestNumber(self, nums):
6         """
7             :type nums: List[int]

```

```

8     :rtype: str
9     """
10    # convert the list of integers to list of strings
11    strs = [str(e) for e in nums]
12    largest_num = ''.join(sorted(strs, key = LargerNumKey))
13    return '0' if largest_num[0] == '0' else largest_num

```

To simply the case, we can use functiontool libraray to convert a function to the above.

```

1 import functools
2 def my_cmp(x, y):
3     if x+y > y+x:
4         return 1
5     elif x+y == y+x:
6         return 0
7     else:
8         return -1
9
10 largest_num = sorted(strs, key = functools.cmp_to_key(my_cmp),
11                      reverse = True)

```

23.7 Summary and Bonus

In this section, compare these sorting, and a potentially a size of an array and how much time they spend comparison. A table and a figure. There is an animation of all the common algorithms¹.

A bubble sort, a selection sort, and an insertion sort are $O(n^2)$ algorithms. A shell sort improves on the insertion sort by sorting incremental sublists. It falls between $O(n)$ and $O(n^2)$. A merge sort is $O(n\log n)$, but requires additional space for the merging process. A quick sort is $O(n\log n)$, but may degrade to $O(n^2)$ if the split points are not near the middle of the list. It does not require additional space.

23.8 LeetCode Problems

Problems

23.1 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

¹<https://www.toptal.com/developers/sorting-algorithms>

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

23.2 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.

23.3 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 = "anagram", t = "nagaram"
Output: true

Example 2:
Input: s = "rat", t = "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?

23.4 Largest Number (179, medium).

23.5 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.*

23.6 148. 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
            cur_node.next = cmp_node
            cur_node = next_node

```

```
31     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
```

```

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 of 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=
8             LargerNumKey))
9         return '0' if largest_num[0] == '0' else
10        largest_num

```

24

Permutation, Combination, and Partition

There are some other combinatorial topics, such as the permutation and combination we have learned how to generate through backtracking technique in Chapter 13. Permutation and combination problems require us to reorganize a given set of items. Other than the backtracking techniques, we show other methods to solve this type of problems.

There are another type of partition, job scheduling.

24.1 Partitions

There are two types of partitions: (1) integer partition and (2) set partitions.

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 seven subsets: $\{5\}$, $\{4, 1\}$, $\{3, 2\}$, $\{3, 1, 1\}$, $\{2, 2, 1\}$, $\{2, 1, 1, 1\}$, and $\{1, 1, 1, 1, 1\}$. Integer partition can be implemented with dynamic programming iteratively.

Set partition is to partition a given set of items into distinct subsets. For example, given a set $1, 2, 3, 4$, it can be partitioned into 1234 , 123 , 4 , $124,3$, 12 , 34 , $12, 3$, 4 , 134 , $2, 13$, 24 , $13, 2$, 4 , 14 , $23,1$, $234,1$, 23 , $4,14,2,3,1,24,3,1,2,34$, and $1,2,3,4$ for parts from 1 to 4. Vertex or edge coloring, and connected components are one of the set partition examples. Set partition is actually a combination problem. We will show how we can do set partition with backtrack and dynamic programming sometimes.

24.1.1 Integer Partition

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, $s > 1$.

24.1.2 Set Partition

24.1.3 Traveling Salesman Problem

Part IX

Math and Geometry

25

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.

25.1 Numbers

25.1.1 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))
```

25.1.2 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 $O(nlogn)$, 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]

```

25.1.3 Combinatorics

1. 611. Valid Triangle Number

25.1 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

```

25.2 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.

25.2.1 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))

```

25.2.2 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:

$$\text{lcm}(a, b) = \frac{a \times b}{\gcd(a, b)} \quad (25.1)$$

25.3 Arithmetic Operations

Because for the computer, it only understands the binary representation as we learned in Bit Manipulation (Chapter 6, 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 $O(n - m)$.

25.2 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

```

25.4 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.

25.5 Linear Algebra

Gaussian Elimination is one of the several ways to find the solution for a system of linear equations.

25.6 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, _
6             in curr])
6             ly, hy = min([y for _, y in curr]), max([y for _, y
7             in curr])
8             size = (rx-lx)*(hy-ly)
9             if size >= ans[0]:
10                 return
11             xs = [lx, rx]
12             ys = [ly, hy]
13             for x, y in curr:
14                 if x not in xs or y not in ys:
15                     return
16
17             if len(curr) == 4:
18                 ans[0] = min(ans[0], size)
19             return
20
21         for i in range(idx, len(points)):
22             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

```

25.7 Miscellaneous Categories

25.7.1 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.

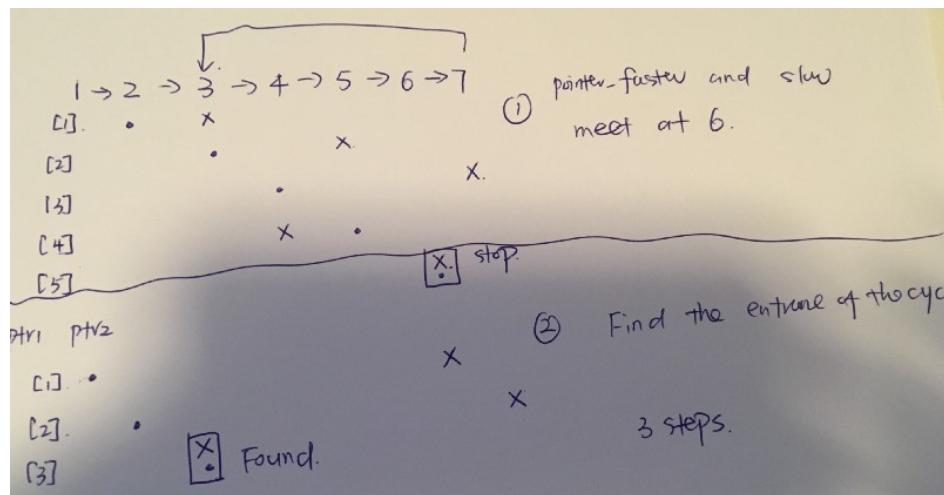


Figure 25.1: Example of floyd's cycle finding

```

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

```

25.8 Exercise

25.8.1 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 X

**Questions by Data
Structures**

26

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 sequence 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)

26.1 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)

26.1.1 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

26.1 **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.

26.2 325. 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
0
9     max_len = 0
10    for idx,n in enumerate(nums):
11        prefix_sum += n
12        # save the set of prefix sum together with the
13        # first index of this value appears.
14        if prefix_sum not in dict:
15            dict[prefix_sum] = idx
16        # track the maximum length so far
17        if prefix_sum-k in dict:
18            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:

- 26.3 **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(
len(nums))]
7
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

```

26.4 674. 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

```

26.5 209. 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

26.6 560. 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

```

- 26.7 **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())

```

26.1.2 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 7.3.4, 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:

26.8 209. 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

```

26.9 713. 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 7.3.4

will fit the scenario and gave $O(n)$ time complexity and $O(N)$ space complexity.

26.10 862. 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 stablize. 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 monoitually increasing like the array with all positive numbers by maintaince 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
13                negative and zero leads to kick out any previous larger
14                or equal value
15                print('pop', P[monoq[-1]])
16                monoq.pop()
17
18            while monoq and Py - P[monoq[0]] >= K: # if one
19                x is considered, no need to consider again (similar to
20                sliding window where we move the first index forward)
21                print('pop', P[monoq[0]])
22                ans = min(ans, y - monoq.popleft())
23                print('append', P[y])
24                monoq.append(y)
25
26
27        return ans if ans < len(A)+1 else -1

```

26.1.3 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
2     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,

26.11 795.Number of Subarrays with Bounded Maximum (medium)

26.12 907. Sum of Subarray Minimums (monotone stack)

26.2 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. 26.1

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)

The screenshot shows a search results page on LeetCode. The search bar at the top contains the word 'subsequence'. Below the search bar, there are filters for 'Difficulty', 'Status', 'Lists', and 'Tags'. The main area displays a table of 26 subsequence-related problems. Each row in the table includes the problem ID, title, solution count, acceptance rate, difficulty level (Medium, Easy, Hard), and frequency. The table has columns for #, Title, Solution, Acceptance, Difficulty, and Frequency.

#	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	Low
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 26.1: 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
    d: d, ad, bd, abd, cd, acd, bcd, abcd = dp[0]+dp[1]+dp[2]+1

```

Thus the recurrence function can be Eq. 26.1.

$$dp[i] = \sum_{j < i} (dp[j]) + 1, S[j]! = S[i] \quad (26.1)$$

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):
11             if S[j] == c:
12                 dp[i] = (dp[i] + dp[j]) % MOD
13
14     return dp[-1]

```

```

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

```

26.2.1 Others

For example, the following question would be used as follow up for question *Longest Continuous Increasing Subsequence*

300. Longest Increasing Subsequence

```

1 Given an unsorted array of integers , find the length of longest
2 increasing subsequence .
3
4 For example ,
5
6 Given [10, 9, 2, 5, 3, 7, 101, 18],
7 The longest increasing subsequence is [2, 3, 7, 101], therefore
8 the length is 4. Note that there may be more than one LIS
9 combination , it is only necessary for you to return the
10 length .
11
12 Your algorithm should run in  $\$O(n^2)$  complexity .
13
14 Follow up: Could you improve it to  $\$O(n\log n)$  time complexity ?
15 \begin{lstlisting}
16 Solution: Compared with the last question , this one loose the
17 restriction that need to be continuous. For this problem , we
18 need to understand it is not going to work with two pointers .
19 It is not a brute-force  $\$O(n^2)$  problem . It is a typical

```

```

combination problem in recursive functions. So, at first , put
the standard combination algorithm code here:
15 \begin{lstlisting}[language = Python]
16 def dfs(temp, idx):
17     rs1t.append(temp[:]) #pass temp[:] with shallow copy
    so that we wont change the result of rs1t when temp is
    changed
18     for i in range(idx, len(nums)):
19         temp.append(nums[i])
        #backtrack
20         dfs(temp, i+1)
21         temp.pop()
22
23
24
25     dfs([],0)
26     return rs1t

```

So, we use the backtracking-combination to enumerate all possible subsequence. The difference is here we do not unconditionally use this $\text{nums}[i]$ in our result, only if $\text{nums}[i] > \text{tail}$, and the final length is the maximum of them all. $T(n) = \max(T(n - 1) + 1, T(n - k) + 1, \dots)$. So, the time complexity is $O(2^n)$. It passed 21/15 test cases with TLE. In this process, we transfer from the combination problem to dynamic programming.

```

1 def lengthOfLIS(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: int
5     """
6
7     max_count = 0
8     if not nums:
9         return 0
10    def backtrackingDFS(idx, tail):
11        if idx == len(nums):
12            return 0
13        length = 0
14        for i in range(idx, len(nums)):
15            if nums[i] > tail:
16                length = max(1 + backtrackingDFS(i + 1, nums[i]))
17            , length)
18        return length
19
20    return backtrackingDFS(0, -maxsize)

```

Now, we know we are doing dynamic programming, if we already know the $\text{ans}(\text{idx})$, meaning the max length from somewhere, we do not need to do it again. With memoization: The time complexity is n subproblem, top-down recursive+memo.

```

1 def lengthOfLIS(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: int

```

```

5     """
6     max_count = 0
7     if not nums:
8         return 0
9     memo =[None for _ in range(len(nums))]
10    def backtrackingDFS(idx, tail):
11        if idx==len(nums):
12            return 0
13        if memo[idx]==None:
14            length = 0
15            for i in range(idx, len(nums)):
16                if nums[i]>tail:
17                    length = max(1+backtrackingDFS(i+1, nums
18 [i]), length)
19            memo[idx]=length
20        return memo[idx]
21
22    return backtrackingDFS(0,-maxsize)

```

Now, we use dynamic programming which its solution can be found in Section ???. And bottom-up iterative. For [10,9,2,5,3], the length array is [1,1,1,2,2], for [4,10,4,3,8,9], we have [1, 2, 1, 1, 2, 3]. To find the rule, $T(0)=1$, idx , $\max(\text{memo}[i])$, $0 \leq i < idx$, $nums[idx] > nums[i]$. Now the time complexity is $O(n^2)$.

state: $f[i]$ record the maximum length of increasing subsequence from 0-i.

function: $f[i]$: choose or not to choose

initialize: $f[0]=1$

```

1 def lengthOfLIS(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: int
5     """
6     max_count = 0
7     if not nums:
8         return 0
9     dp=[0 for _ in range(len(nums))]
10    dp[0]=1
11    maxans =1
12    for idx in range(1, len(nums)): #current combine this to
13        this subsequence, 10 to [], 9 to [10]
14        pre_max=0
15        for i in range(0, idx):
16            if nums[idx]>nums[i]:
17                pre_max=max(pre_max, dp[i])
18            dp[idx]=pre_max+1
19            maxans=max(maxans, dp[idx])
20
21    print(dp)
22    return maxans

```

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

```

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 ,
4 , 7] and [1 , 3 , 5 , 7].
6
7 Example  2:
8 Input:  [2 ,2 ,2 ,2 ,2]
9 Output: 5

```

```

10 Explanation: The length of longest continuous increasing
   subsequence is 1, and there are 5 subsequences' length is 1,
   so output 5.
11 \textit{Note: Length of the given array will be not exceed 2000
   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
16                    counts[idx] = counts[i] #change the
17                    count
18                elif lengths[i] + 1 == lengths[idx]: #if it
19                    is a tie
20                    counts[idx] += counts[i] #increase the
21                    current count by count[i]
22
23    longest = max(lengths)
24    print(counts)
25    print(lengths)
26    return sum(c for i, c in enumerate(counts) if lengths[i]
27               == longest)
28

```

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 .
10

```

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

```

```

12         maxlen = max(maxlen, last - first + 1)
13     return maxlen

```

26.3 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 13.1.2. 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.

26.3.1 Combination

The solution of this section is heavily correlated to Section 13.1.2. 78. Subsets

```

1 Given a set of distinct integers , nums, return all possible
2   subsets (the power set).
3
4 Note: The solution set must not contain duplicate subsets.
5
6 Example:
7
8 Input: nums = [1,2,3]
9 Output:
10 [
11   [3],
12   [1],
13   [2],
14   [1,2,3],
15   [1,3],
16   [2,3],
17   [1,2],
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
15        ans.append(curr)
16        if d == k: #the length is satisfied
17            return
18        for i in range(s, n):
19            curr.append(nums[i])
20            C_n_k(d+1, k, i+1, curr[:], ans) # i+1 because no
21            repeat , make sure use deep copy curr [:]
22            curr.pop()
23
24    ans = []
25    C_n_k(0, k, 0, [], ans)
26    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 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
5 [1, 2]
6 l = 3, for 3, []+[3], [1]+[3], [2]+[3], [1, 2]+[3], -> [3], [1,
7 3], [2, 3], [1, 2, 3], get powerset of [1, 2, 3]
8 l = 4, for 4, []+[4]; [1]+[4]; [2]+[4], [1, 2] +[4]; [3]+[4],
9 [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 []
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

```

90. Subsets II

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         nums.sort()
7         result = [[]] #use two dimensional, which already have
8         one element
9         for num in nums:
10            new_results = []
11            for r in result:
12                print(r)
13                new_results.append(r + [num])
14            for rst in new_results:
15                if rst not in result: # check the repetitive
16                    result.append(rst)
17
18        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     if not nums:
8         return [[]]
9     res = []
10    dic = collections.Counter(nums)
11    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 Example :
4
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 13.1.2.

```

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

```

26.3.2 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, 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?

```

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
   index is same or larger than before)
3 [
4   [2,2,2,2],
5   [2,3,3],
6   [3,5]
7 ]
8 377. Combination Sum IV, here we have no ordering( each time the
   start index is the same as before). Try all element.
9 [
10  [2,2,2,2],
11  [2,3,3],
12 * [3,3,2]
13 * [3,2,3]
14  [3,5],
15 * [5,3]
16 ]

```

```

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 ??.

26.3.3 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
5 Because nums[0] + nums[1] = 2 + 7 = 9,
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
6             not repeat
7                 continue
8         l, r = i+1, len(nums)-1
9         while l < r:
10            s = nums[i] + nums[l] + nums[r]
11            if s < 0:
12                l +=1
13            elif s > 0:
14                r -= 1
15            else:
16                res.append((nums[i], nums[l], nums[r]))
17                l+=1
18                r-=1
19
20             #after the first run, then check duplicate
21             example.
22             while l < r and nums[l] == nums[l-1]:
23                 l += 1
24                 while l < r and nums[r] == nums[r+1]:
25                     r -= 1
26
27     return res

```

Use hashmap:

```

1 def threeSum(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: List[List[int]]
5     """
6
7     res = []
8     nums=sorted(nums)
9     if not nums:
10        return []
11     if nums[-1]<0 or nums[0]>0:
12        return []
13     end_position = len(nums)-2
14     dic_nums={}
15     for i in xrange(1, len(nums)):
16         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
    continue
if target<nums[i]: #if the target is smaller than
this, we can not find them on the right side
    break
for j in range(i+1,len(nums)): #this is to avoid
repeat
    if j>i+1 and nums[j]==nums[j-1]:
        continue
    complement =target - nums[j]
    if complement<nums[j]: #if the left numbers are
bigger than the complement, no need to keep searching
        break
    if complement in dic_nums and dic_nums[
complement]>j: #need to make sure the complement is bigger
than nums[j]
        res.append([nums[i],nums[j],complement])
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[l] == 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 [ 1 ] + nums [ r ]
9         if s == target :
10            results . append ( result + [ nums [ 1 ] , nums [ r
11        ] ] )
12        l += 1
13        r -= 1
14        while l < r and nums [ 1 ] == nums [ l - 1 ] :
15            l += 1
16        while l < r and nums [ r ] == nums [ r + 1 ] :
17            r -= 1
18        elif s < target :
19            l += 1
20        else :
21            r -= 1
22    else : # recursively reduce N
23        for i in range ( len ( nums ) - N + 1 ) :
24            if i == 0 or ( i > 0 and nums [ i - 1 ] != nums [ i
25        ] ) :
26                findNsum ( nums [ i + 1 : ] , target - nums [ i ] , N
27                - 1 , result + [ nums [ i ] ] , results ) #reduce nums size , reduce
28                target , save result
29
30 results = []
31 findNsum ( sorted ( nums ) , target , 4 , [] , results )
32 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 13.1.2 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.

26.3.4 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     [1,2,3],
8     [1,3,2],
9     [2,1,3],
10    [2,3,1],
11    [3,1,2],
12    [3,2,1]
13 ]
```

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 ")"(" -> []

```

26.4 Merge and Partition

26.4.1 Merge Lists

We can use divide and conquer (see the merge sort) and the priority queue.

26.4.2 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

26.5 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

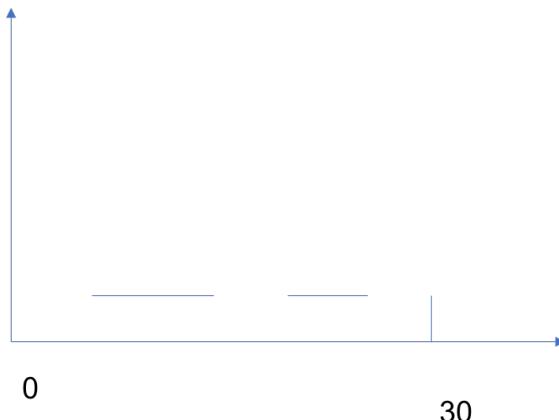


Figure 26.2: Interval questions

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 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
25         for i in range(s+1,e+1):
26             votes[i]+=1
27             num_rooms = max(num_rooms, votes[i])
28     return num_rooms

```

26.5.1 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:

```

1 def minMeetingRooms(self, intervals):
2     if not intervals:
3         return 0
4     #solution 2
5     slots = []

```

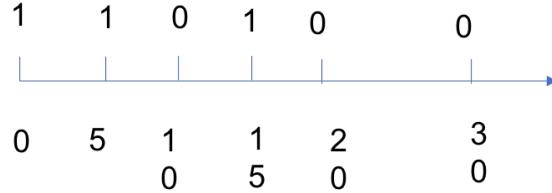


Figure 26.3: One-dimensional Sweep Line

```

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

```

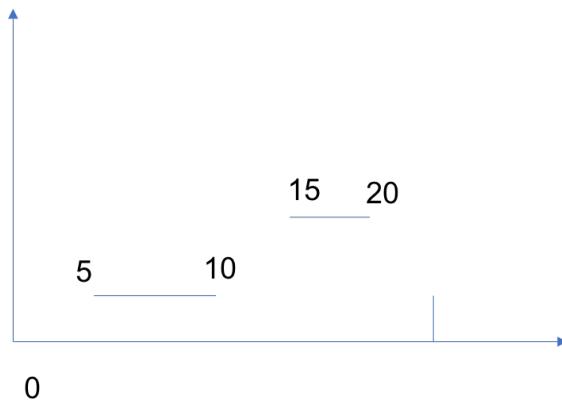
Min-heap Implementation

Figure 26.4: 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. 26.4, 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 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 };

```

26.5.2 LeetCode Problems

1. **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.

26.6 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     nums1.sort()
8     nums2.sort()
9     r = set()
10    i, j = 0, 0
11    while i < len(nums1) and j < len(nums2):
12        if nums1[i] < nums2[j]:
13            i += 1
14        elif nums1[i] > nums2[j]:
15            j += 1
16        else:
17            r.add(nums1[i])
18            i += 1
19            j += 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?

26.7 Miscellaneous Questions

- 26.13 **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 $O(n^2)$.

26.8 Exercises

26.8.1 Subsequence with (DP)

1. 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
13     for key,item in count.items(): #to visit the key:
14         item in the counter
15         if count[key+1]: #because the list is sorted,
16             so we only need to check key+1
17             maxLen = max(maxLen,item+count[key+1])
18
19             # if count[key-1]:
20             #     maxLen=max(maxLen, item+count[key-1])
21
22     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
4     "cdc"),
5 because "aba" is a subsequence of "aba",
6 but not a subsequence of any other strings in the group of
7     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     #what if we only have one char
14     # for char1 in chars.keys():
15     #     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, str=`“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                   ()):
24                 newCounterF=count .copy()
25                 newCounterF[s[start]]-=1
26                 newCounterB=count .copy()
27                 newCounterB[s[end]]-=1
28                 #print(newsF,newsB)
29                 memo[start][end]= max(cut(start+1, end,
30                     newCounterF), cut(start , end-1, newCounterB))
31             else:
32                 memo[start][end] = end-start+1
33         return memo[start][end]
34     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)

```

26.8.2 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

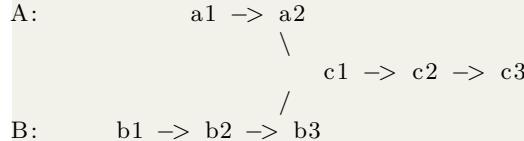
```

26.8.3 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.

27

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.

27.1 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.

27.1 708. 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.

546 27. LINKED LIST, STACK, QUEUE, AND HEAP QUESTIONS (12%)

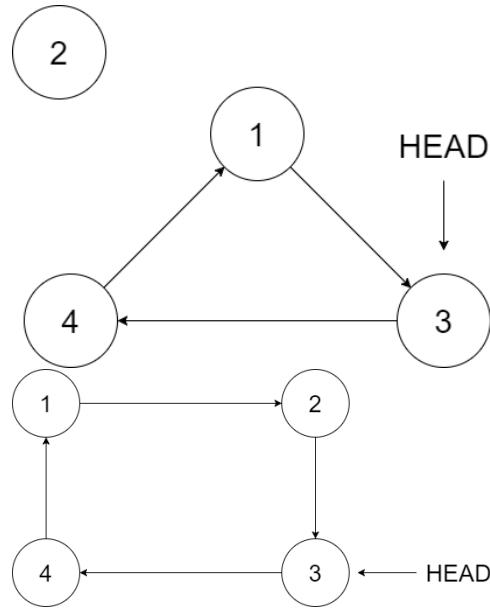


Figure 27.1: Example of insertion in circular list

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
    return head
  
```

27.2 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),

27.2.1 Implementing Queue and Stack

27.2 622. 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

```

548 27. 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
```

27.3 641. Design Circular Deque (medium).

Solution: Doubly linked List with Predefined size

27.2.2 Solving Problems Using Queue

Use as a Buffer

27.4 346. 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 :

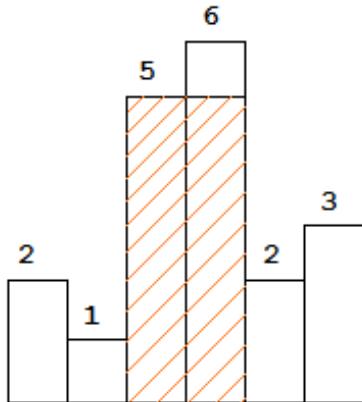


Figure 27.2: Histogram

```
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)
```

27.2.3 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

550 27. 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
31     #handle the left stack
32     while stack[-1]!=-1:
33         idx = stack.pop()
34         v = heights[idx]
35         maxsize=max(maxsize, (len(heights)-stack[-1]-1)
36 *v)
37
38     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
7     if not matrix:
8         return 0
9     if len(matrix[0])==0:
10        return 0
11    row, col = len(matrix), len(matrix[0])
12
13    def check(x,y,w,h):
14        #check the last col
15        for i in range(x, x+h): #change row
16            if matrix[i][y+w-1]=='0':
17                return 0
18        for j in range(y, y+w): #change col
19            if matrix[x+h-1][j]=='0':
20                return 0
21        return w*h
22    maxsize = 0
23    for i in range(row):
24        for j in range(col): #start point i,j
25            if matrix[i][j]=='0':
26                continue
27                for h in range(1, row-i+1): #decide the
28                    for w in range(1, col-j+1):
29                        maxsize = max(maxsize, check(i, j, w, h))
30
31    return maxsize

```

552 27. 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

554 27. 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

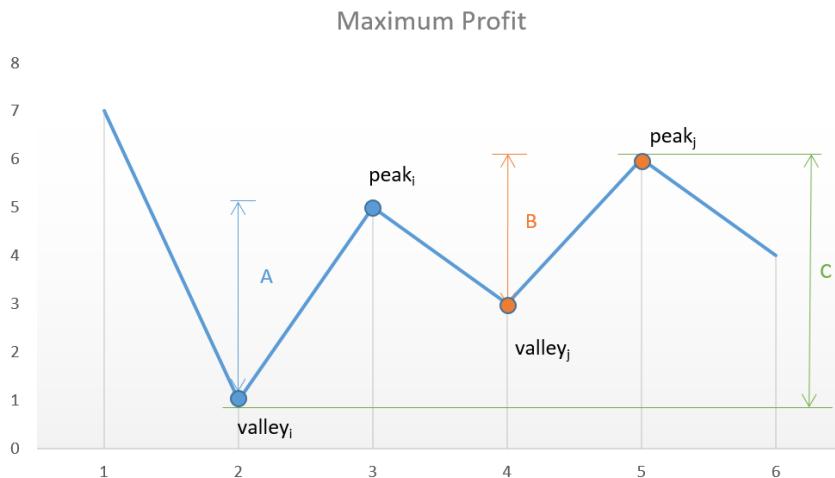


Figure 27.3: Track the peaks and valleys

interest are the consecutive valleys and peaks.

Mathematically speaking:

$$\text{TotalProfit} = \sum_i (\text{height}(\text{peak}_i) - \text{height}(\text{valley}_i)) \quad (27.1)$$

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

556 27. 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 }
```

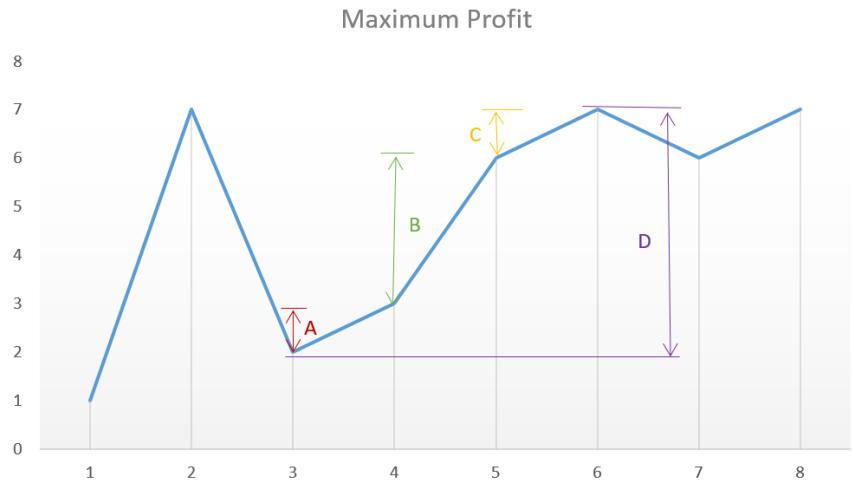


Figure 27.4: profit graph

27.3 Heap and Priority Queue

27.5 621. 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,

558 27. 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. 27.5, 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)

```

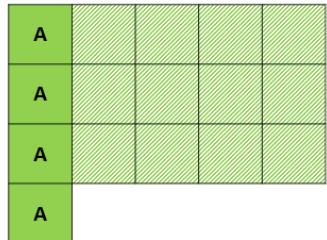


Figure 1

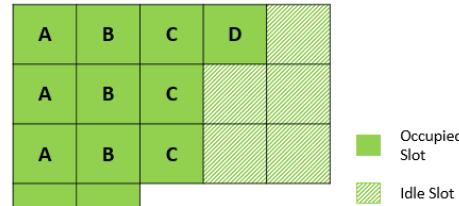


Figure 2

Figure 27.5: 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)

```

560 27. *LINKED LIST, STACK, QUEUE, AND HEAP QUESTIONS (12%)*

28

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 28.1. (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 28.3

- 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 28.5 (2) If we care the order matters (identical to pattern), we use KMP. The problems of this type is listed in Section 28.6.

28.1 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)

28.2 String Expression

1. 8. String to Integer (atoi) (medium)

28.3 Advanced Single String

For hard problem, reconstruct the problem to another so that it can be resolved by an algorithm that you know.

28.3.1 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.

28.1 **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
15    return True

```

28.2 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

```

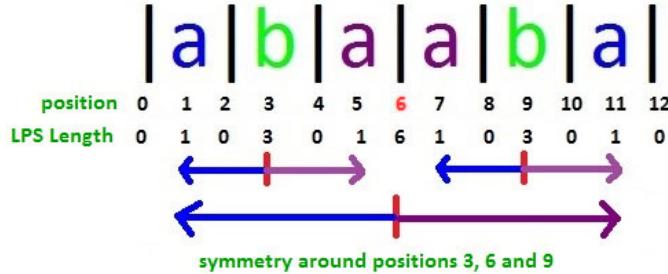


Figure 28.1: LPS length at each position for palindrome.

```
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))
```

28.3 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, 1], [0, 0, 0, 0, 1]]  
[[1, 2, 3, 3, 0], [0, 1, 2, 2, 3], [0, 0, 1, 1, 3], [0, 0, 0, 1, 1], [0, 0, 0, 0, 1]]  
[[1, 2, 3, 3, 4], [0, 1, 2, 2, 3], [0, 0, 1, 1, 3], [0, 0, 0, 1, 1], [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]

```

28.3.2 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:

$$\frac{(a + \frac{b}{c})_a * (d - e)_d}{a} + \frac{((f - g)_f - (h - i)_h)}{f} + \frac{(j - k)_j}{j} \quad (28.1)$$

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.

28.4 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
```

28.5 **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

```

28.6 227. Basic Calculator II (exercise)

28.3.3 Others

Possible methods: two pointers, one loop+two pointers

28.4 Exact Matching: Sliding Window and KMP

Exact Pattern Matching

14. Longest Common Prefix (easy)

28.5 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

```

28.6 Exact Matching

28.6.1 Longest Common Subsequence

28.7 Exercise

28.7.1 Palindrome

28.1 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"

29

Tree Questions(10%)

Review We have learned the tree data structure and the traversal inside of the tree from Chapter 8 and Chapter 15. And it is important to remind everyone the core and fundamental methods for solving tree related problems is the **tree traversal** and **divide-and-conquer**. We are expecting problems related to the properties of trees. And efficiency is important.

29.1 Core Principle

When we first go for interviews, we may find tree and graph problems intimidating and challenging to solve within 40 mins normal interview window. This might because of our neglect of the concept of Divide and Conquer or due to the recursion occurrence in tree-related problems. However, at this point, we have already studied the concepts of various trees, divide and conquer, and solved quite a large amount of related questions in previous chapters in this part. We will find out studying this chapter can be real easy compared with the Dynamic programming questions because the consistent principle to solve tree questions. The principle is to solve problems within **tree traversal**, either recursively or iteratively, in either of two ways:

1. **Top-down Searching:** Tree traversal and with visited nodes information as parameters to be passed to its subtree. The result will be returned from leaf node or empty node, or node that satisfy a certain condition. This is just an extension of the graph search, either BFS or DFS, with recorded path information. This usually requires None return from the recursion function, but instead always require a global data structure and a local data structure to track the final answer and the current path information.

```

1 def treeTraversal(root, tmp_result):
2     if node is empty or node is a leaf node:
3         collect the result or return the final result
4     construct: the previous temp result using the current
      node
5     treeTraversal(root.left, constructed_tmp_result)
6     treeTraversal(root.right, constructed_tmp_result)

```

2. **Bottom-up Divide and Conquer:** Due to the special structure of tree, a tree is naturally divided into two halves: left subtree and right subtree. Therefore, we can enforce the Divide and Conquer, to assign two “agents” to obtain the result for its subproblems, and back to current node, we “merge” the results of the subtree to gain the result for current node. This also requires us to define the return value for edge cases: normally would be empty node and/or leaves.

```

1 def treeTraversalDivideConquer(root):
2     if node is empty or node is a leaf node:
3         return base result
4     # divide
5     left result = treeTraversalDivideConquer(root.left)
6     right result = treeTraversalDivideConquer(root.right)
7
8     # conquer
9     merge the left and right result with the current node
10    return merged result of current node

```

The difficulty of these problems are decided by the merge operation, and how many different variables we need to return to decide the next merge operation. However, if we don’t like using the recursive function, we can use levelorder traversal implemented with Queue.

Summary of Recurrence Relation Due to the fact that divide-and-conquer is inherent to the recursive structure of trees and root-children relation; a binary tree of size n with root will be divided into three parts: left-subtree, root, right-subtree. The problem of n is thus divided into $d(r) = (d(r.left), d(r.right), r)$, and there has no overlapping between these three parts.

Let us conclude some recurrence relation function to the different property of trees, here we use binary tree for the demonstration purpose. In a given tree t , and its left and right subtree are denoted by t_l and t_r , respectively.

1. Count the size of a tree: let $|t|$ to be the number of nodes in the tree, and an empty node will have size 0. the recursive relation will be:

$$|t| = |t_l| + |t_r| + 1 \quad (29.1)$$

2. Obtain the height of the tree: let $h(t)$ be the height of node t , and a leaf node will have height 0, and thus an empty node will have -1 . We have height the recursive relation is:

$$h(t) = 1 + \max(h(t_l, t_r)) \quad (29.2)$$

3.

N-ary Tree, Binary tree, and Binary Search tree are the most popular type of questions among interviews. They each take nearly half and half of all the tree questions. We would rarely come into the Segment Tree or Trie, but if you have extra time it will help you learn more if you would study these two types too.

29.2 N-ary Tree (40%)

We classify the binary tree related questions as:

1. Tree Traversal;
2. Tree Property: Depth, Height, and Diameter
3. Tree Advanced Property: LCA
4. Tree Path

29.2.1 Tree Traversal

The problems appearing in this section has mostly been solved in tree traversal section, thus we only list the problems here.

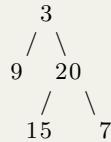
1. 144. Binary Tree Preorder Traversal
2. 94. Binary Tree Inorder Traversal
3. 145. Binary Tree Postorder Traversal
4. 589. N-ary Tree Preorder Traversal
5. 590. N-ary Tree Postorder Traversal
6. 429. N-ary Tree Level Order Traversal
7. 103. Binary Tree Zigzag Level Order Traversal(medium)
8. 105. Construct Binary Tree from Preorder and Inorder Traversal

29.1 103. Binary Tree Zigzag Level Order Traversal (medium).

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:

Given binary tree [3,9,20,null,null,15,7],



return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

Solution: BFS level order traversal. We use a variable to track the level of the current queue, and if its even, then we add the result in the original order, otherwise, use the reversed order:

```

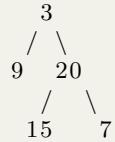
1 def zigzagLevelOrder(self, root):
2     """
3         :type root: TreeNode
4         :rtype: List[List[int]]
5     """
6     if root is None:
7         return []
8     q = [root]
9     i = 0
10    ans = []
11    while q:
12        tmp = []
13        tmpAns = []
14        for node in q:
15            tmpAns.append(node.val)
16            if node.left:
17                tmp.append(node.left)
18            if node.right:
19                tmp.append(node.right)
20        q = tmp
21        if i % 2 == 0:
22            ans += [tmpAns]
23        else:
24            ans += [tmpAns[::-1]]
25        i += 1
26    return ans
  
```

29.2 105. Construct Binary Tree from Preorder and Inorder Traversal.

Given preorder and inorder traversal of a tree, construct the binary tree. Note: You may assume that duplicates do not exist in the tree.

For example, given preorder = [3, 9, 20, 15, 7], inorder = [9, 3, 15, 20, 7]

Return the following binary tree:



Solution: the feature of tree traversal. The inorder traversal puts the nodes from the left subtree on the left side of root, and the nodes from the right subtree on the right side of the root. While the preorder puts the root at the first place, followed by the left nodes and right nodes. Thus we can find the root node from the preorder, and then use the inorder list to find the root node, and cut the list into two parts: left nodes and right nodes. We use divide and conquer, and do such operation recursively till the preorder and inorder list is empty.

```

1 def buildTree(self, preorder, inorder):
2     """
3         :type preorder: List[int]
4         :type inorder: List[int]
5         :rtype: TreeNode
6     """
7     #first to decide the root
8     def helper(preorder,inorder):
9         if not preorder or not inorder:
10            return None
11
12        cur_val = preorder[0]
13        node = TreeNode(cur_val)
14        #divide: now cut the lists into two halves
15        leftinorder,rightinorder = [],[]
16        bLeft=True
17        for e in inorder:
18            if e==cur_val:
19                bLeft=False #switch to the right side
20                continue
21            if bLeft:
22                leftinorder.append(e)
23            else:
24                rightinorder.append(e)
25        leftset, rightset = set(leftinorder),set(
26            rightinorder)
27        leftpreorder, rightpreorder = [],[]
  
```

```

27     for e in preorder[1:]:
28         if e in leftset:
29             leftpreorder.append(e)
30         else:
31             rightpreorder.append(e)
32
33     #conquer
34     node.left=helper(leftpreorder, leftinorder)
35     node.right= helper(rightpreorder, rightinorder)
36     return node
37 return helper(preorder, inorder)

```

However, the previous code has problem as 203 / 203 test cases passed.
 Status: Memory Limit Exceeded. So instead of passing new array, I use index.

```

1 def buildTree(self, preorder, inorder):
2     """
3     :type preorder: List[int]
4     :type inorder: List[int]
5     :rtype: TreeNode
6     """
7     #first to decide the root
8     def helper(pre_l, pre_r, in_l, in_r): #[pre_l, pre_r)
9         if pre_l>=pre_r or in_l>=in_r:
10             return None
11
12         cur_val = preorder[pre_l]
13         node = TreeNode(cur_val)
14         #divide: now cut the lists into two halves
15         leftinorder = set()
16         inorder_index = -1
17         for i in range(in_l, in_r):
18             if inorder[i]==cur_val:
19                 inorder_index = i
20                 break
21             leftinorder.add(inorder[i])
22         #when leftset is empty
23         new_pre_r=pre_l
24         for i in range(pre_l+1,pre_r):
25             if preorder[i] in leftinorder:
26                 new_pre_r = i
27             else:
28                 break
29         new_pre_r+=1
30
31     #conquer
32     node.left=helper(pre_l+1, new_pre_r, in_l,
33     inorder_index)
34     node.right= helper(new_pre_r, pre_r,
35     inorder_index+1, in_r)
36     return node
37     if not preorder or not inorder:
38         return None

```

```
37     return helper(0, len(preorder), 0, len(inorder))
```

29.2.2 Depth/Height/Diameter

In this section, focus on the property related problems of binary tree: including depth, height and diameter. We can be asked to validate balanced binary tree, or the maximum/minimum of these values. The solution is tree traversal along with some operations along can be used to solve this type of problems.

1. 111. Minimum Depth of Binary Tree (Easy)
2. 110. Balanced Binary Tree(Easy)
3. 543. Diameter of Binary Tree (Easy)
4. 559. Maximum Depth of N-ary Tree (Easy) (Exercise)
5. 104. Maximum Depth of Binary Tree (Exercise)

29.3 Minimum Depth of Binary Tree (L111, Easy). Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node. *Note: A leaf is a node with no children.*

```
1 Example:
2
3 Given binary tree [3,9,20,null,null,15,7],
4
5      3
6      / \
7      9   20
8      /   \
9      15   7
10
11 return its minimum depth = 2.
```

Solution 1: Level-Order Iterative. For the minumum path, we can traverse the tree level-by-level and once we encounter the first leaf node, this would be the minimum depth and we return from here and has no need to finish traversing the whole tree. The worst time complexity is $O(n)$ and with $O(n)$ space.

```
1 def minDepth(self, root):
2     if root is None:
3         return 0
4     q = [root]
5     d = 0
6     while q:
7         d += 1
8         for node in q:
```

```

9         if not node.left and not node.right: #a leaf
10        return d
11
12        q = [neigbor for n in q for neigbor in [n.left, n.
13          right] if neigbor]
13        return d

```

Solution 2: DFS + Divide and Conquer. In this problem, we can still use a DFS based traversal. However, in this solution, without iterating the whole tree we would not get the minimum depth. So, it might take bit longer time. And, this takes $O(h)$ stack space.

```

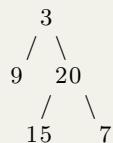
1 def minDepth(self, root):
2     if not root:
3         return 0
4     if not root.left and not root.right: # only leaves will
5         have 1
6         return 1
7     ans = sys.maxsize
8     if root.left:
9         ans = min(ans, self.minDepth(root.left))
10    if root.right:
11        ans = min(ans, self.minDepth(root.right))
11    return ans+1

```

29.4 110. Balanced Binary Tree(L110, Easy). Given a binary tree, determine if it is height-balanced. For this problem, a height-balanced binary tree is defined as: *a binary tree in which the height of the two subtrees of every node never differ by more than 1.* (LeetCode used depth however, it should be the height)

Example 1:

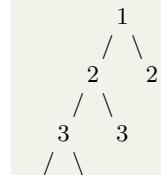
Given the following tree [3,9,20,null,null,15,7]:



Return true.

Example 2:

Given the following tree [1,2,2,3,3,null,null,4,4]:



```
4    4
```

```
Return false.
```

Solution 1: Bottom-up DFS+Divide and conquer with height as return. First, because the height of a tree is defined as the number of edges on the *longest path* from node to a leaf. And a leaf will have a height of 0. Thus, for the DFS traversal, we need to return 0 for the leaf node, and for an empty node, we use -1 (for leaf node, we have $\max(-1, -1) + 1 = 0$). In this process, we just need to check if the left subtree or the right subtree is already unbalanced which we use -2 to denote, or the difference of the height of the two subtrees is more than 1.

```
1 def isBalanced(self, root):
2     """
3     :type root: TreeNode
4     :rtype: bool
5     """
6     def dfsHeight(root):
7         if not root:
8             return -1
9         lh = dfsHeight(root.left)
10        rh = dfsHeight(root.right)
11        if lh == -2 or rh == -2 or abs(lh-rh) > 1:
12            return -2
13        return max(lh, rh)+1
14    return dfsHeight(root) != -2
```

29.5 543. Diameter of Binary Tree (Easy). Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the **longest** path between any two nodes in a tree. Note: The length of path between two nodes is represented by the number of edges between them. This path may or may not pass through the root.

Example:

Given a binary tree



Return 3, which is the length of the path [4,2,1,3] or [5,2,1,3].

Solution: Height of the tree with global variable to track the diameter. For node 2, the height should be 1, and the length of path from 4 to 5 is 2, which is sum of the height of 4, 5 and two edges. Thus,

we use rootToLeaf to track the height of the subtree. Meanwhile, for each node, we use a global variable to track the maximum path that pass through this node, which we can get from the height of the left subtree and right subtree.

```

1 def diameterOfBinaryTree(self, root):
2     """
3         :type root: TreeNode
4         :rtype: int
5     """
6     # this is the longest path from any to any
7
8     def rootToAny(root, ans):
9         if not root:
10             return -1
11         left = rootToAny(root.left, ans)
12         right = rootToAny(root.right, ans)
13         ans[0] = max(ans[0], left+right+2)
14         return max(left, right) + 1
15     ans = [0]
16     rootToAny(root, ans)
17     return ans[0]
```

29.2.3 Paths

In this section, we mainly solve path related problems. As we mentioned in Chapter ??, there are three types of path depending on the starting and ending node type of the path. We might be asked to get minimum/maximum/each path sum/ path length from these three cases: 1) **root-to-leaf**, 2) **Root-to-Any** node, 3) **Any-node to-Any** node.

Also, maximum or minimum questions is more difficult than the exact path sum, because sometimes when there are negative values in the tree, it makes the situation harder.

We normally have two ways to solve these problems. One is using DFS traverse and use global variable and current path variable in the parameters of the recursive function to track the path and collect the results.

The second way is DFS and Divide and Conquer, we treat each node as a root tree, we return its result, and for a node, after we get result of left and right subtree, we merge the result.

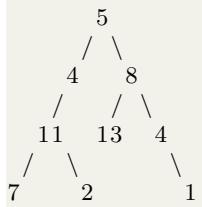
Root to Leaf Path

1. 112. Path Sum (Easy)
2. 113. Path Sum II (easy)
3. 129. Sum Root to Leaf Numbers (Medium)
4. 257. Binary Tree Paths (Easy, exer)

- 29.6 **112. Path Sum (Easy).** Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Note: A leaf is a node with no children.

Example :

Given the below binary tree and sum = 22,



return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

Solution: Tree Traversal, Leaf Node as Base Case. Here we are asked the root-to-leaf path sum, we just need to traverse the tree and use the remaining sum after minusing the value of current node to visit its subtree. At the leaf node, if the remaining sum is equal to the node's value, we return True, otherwise False is returned. Time complexity is $O(n)$.

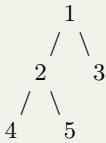
```

1 def hasPathSum( self , root , sum):
2     """
3         :type root: TreeNode
4         :type sum: int
5         :rtype: bool
6     """
7     if root is None: # this is for empty tree
8         return False
9     if root.left is None and root.right is None: # a leaf
10        as base case
11            return True if sum == root.val else False
12
13     left = self.hasPathSum( root.left , sum-root.val)
14     if left:
15         return True
16     right = self.hasPathSum( root.right , sum-root.val)
17     if right:
18         return True
19     return False
  
```

- 29.7 **129. Sum Root to Leaf Numbers (Medium).** Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number. An example is the root-to-leaf path 1->2->3 which represents the number 123. Find the total sum of all root-to-leaf numbers. Note: A leaf is a node with no children.

Example :

Input: [1, 2, 3, 4, 5]



Output: 262

Explanation :

The root-to-leaf path 1->2->4 represents the number 124.

The root-to-leaf path 1->2->5 represents the number 125.

The root-to-leaf path 1->3 represents the number 13.

Therefore , sum = 124 + 125 + 13 = 262.

Solution 1: Divide and Conquer. In divide and conquer solution, we treat each child as a root, for node 4 and 5, they return 4 and 5. For node 2, it should get 24+25, in order to construct this value, the recursive function should return the value of its tree and the path length (number of nodes) of current node to all of its leaf nodes. Therefore for node 2, it has two paths: one with 4, and path length is 1, we get $2 * 10^{len} + \text{left}$, and the same for the right side. The return

```

1 def sumNumbers(self, root):
2     """
3     :type root: TreeNode
4     :rtype: int
5     """
6     if not root:
7         return 0
8     ans, _ = self.sumHelper(root)
9     return ans
10 def sumHelper(self, root):
11     if not root:
12         return (0, [])
13     if root.left is None and root.right is None:
14         return (root.val, [1]) # val and depth
15     left, ld = self.sumHelper(root.left)
16     right, rd = self.sumHelper(root.right)
17     # process: sum over the results till this subtree
18     ans = left+right
19     new_d = []
20     for d in ld+rd:
21         new_d.append(d+1)
22     ans += root.val*10**d
23     return (ans, new_d)
  
```

Solution 2: DFS and Parameter Tracker. We can also construct the value from top-down, we simply record the path in the tree traversal, and at the end, we simply convert the result to the final answer.

```

1 def sumNumbers(self, root):
2     """
  
```

```

3     :type root: TreeNode
4     :rtype: int
5     """
6     my_sum = []
7
8     self.dfs(root, "", my_sum)
9
10    res = 0
11
12    for ele in my_sum:
13        res += int(ele) # convert a list to an int?
14
15    return res
16
17
18    def dfs(self, node, routine, my_sum):
19        if not node:
20            return
21
22        routine = routine + str(node.val)
23        if not node.left and not node.right:
24            my_sum.append(routine)
25
26        self.dfs(node.left, routine, my_sum)
27        self.dfs(node.right, routine, my_sum)

```

Root to Any Node Path

Any to Any Node Path

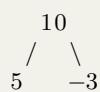
In this subsection, we need a concept called Dual Recursive Function.

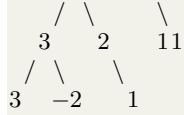
1. 437. Path Sum III (medium)
2. 124. Binary Tree Maximum Path Sum (hard)
3. 543. Diameter of Binary Tree (Easy, put in exercise)

29.8 437. Path Sum III You are given a binary tree in which each node contains an integer value. Find the number of paths that sum to a given value. The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes). The tree has no more than 1,000 nodes and the values are in the range -1,000,000 to 1,000,000.

Example :

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8
```





Return 3. The paths that sum to 8 are:

1. $5 \rightarrow 3$
2. $5 \rightarrow 2 \rightarrow 1$
3. $-3 \rightarrow 11$

Solution 1: Dual Recurrence with Divide and Conquer. In this problem, it is from any to any node, it is equivalent to finding the root->any with sum for all the nodes in the binary tree. We first write a function for root to any. The complexity is $O(n)$.

```

1 def rootToAny(self, root, sum):
2     if root is None:
3         return 0
4     # collect result at any node
5     sum -= root.val
6     count = 0
7     if sum == 0:
8         count += 1
9     return count + self.rootToAny(root.left, sum) + self.rootToAny(root.right, sum)
  
```

However, to get the sum of any to any path (downwards), for each node, we treat it as root node, and call rootToAny, to get satisfactory total paths starts from current node, and we divide the remaining tasks (starting from any other nodes to its left and right subtree). Thus the time complexity if $O(n^2)$. n subproblems and each takes $O(n)$ time.

```

1 '''first recursion: we traverse the tree and use any
2     node as root, and call rootToAny to get its paths'''
3 def pathSum(self, root, sum):
4     if not root:
5         return 0
6     return self.rootToAny(root, sum) + self.pathSum(root.left, sum) + self.pathSum(root.right, sum)
  
```

Solution 2: Optimization with Prefix Sum. The above solution has large amount of recomputation. This is similar in being in an array: we need to set two pointers, one for subarray start and another for the end. We can use prefix sum to decrease the time complexity to $O(n)$. The sum from n_1 to n_2 is $P[n_2] - P[n_1] = \text{sum}$, thus, we need to check $P[n_1]$, which equals to $P[n_2] - \text{sum}$ at each node. To deal with case: [0,0], $\text{sum} = 0$, we need to add 0:1 into the hashmap. Another difference is: in the tree we are using DFS traversal, for a given node, when we finish visit its left subtree and right subtree, and return to

its parent level, we need to reset the hashmap. So, this is DFS with backtracking too.

```

1 def anyToAnyPreSum(self, root, sum, curr, ans, preSum):
2     if root is None:
3         return
4     # process
5     curr += root.val
6     ans[0] += preSum[curr-sum]
7     preSum[curr] += 1
8     self.anyToAnyPreSum(root.left, sum, curr, ans, preSum)
9     self.anyToAnyPreSum(root.right, sum, curr, ans, preSum)
10    preSum[curr] -= 1 #backtrack to current state
11    return
12
13 def pathSum(self, root, sum):
14     if not root:
15         return 0
16     ans = [0]
17     preSum = collections.defaultdict(int)
18     preSum[0] = 1
19     self.anyToAnyPreSum(root, sum, 0, ans, preSum)
20     return ans[0]
```

29.9 124. Binary Tree Maximum Path Sum (hard). Given a non-empty binary tree, find the maximum path sum. For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain at least one node and **does not need to go through the root**.

Example 1:

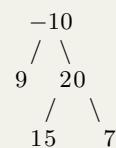
Input: [1,2,3]



Output: 6

Example 2:

Input: [-10,9,20,null,null,15,7]



Output: 42

Solution 1: Dual Recurrence: Before we head over to the optimized solution, first to understand the question. The question can be rephrased as: for each node, find the largest path sum that goes through this node (the path must contain at least one node thus the current node is the one it must include) which is being treated as a root, that is the largest left path sum and the largest right path sum, $\max(\text{ans}[0], \max(\text{left}, 0) + \max(\text{right}, 0) + \text{root.val})$. At first, we gain the max path sum from the root to any node, which we implement in the function `maxRootToAny`. And at the main function, we call `maxRootToAny` for left and right subtree, then merge the result, then we traverse to the left branch and right branch to do those things too. This is a straightforward dual recurrence. With time complexity $O(n^2)$.

```

1 def maxRootToAny(self, root):
2     if root is None:
3         return 0
4     left = self.maxRootToAny(root.left)
5     right = self.maxRootToAny(root.right)
6     # conquer: the current node
7     return root.val + max(0, max(left, right)) #if the left
8     and right are both negative, we get rid of it
9
10    def maxPathSum(self, root):
11        """
12            :type root: TreeNode
13            :rtype: int
14        """
15
16        if root is None:
17            return 0
18        def helper(root, ans):
19            if root is None:
20                return
21            left = self.maxRootToAny(root.left)
22            right = self.maxRootToAny(root.right)
23            ans[0] = max(ans[0], max(left, 0) + max(right, 0) + root
24            .val)
25            helper(root.left, ans)
26            helper(root.right, ans)
27        return
28        ans = [-sys.maxsize]
29        helper(root, ans)
30        return ans[0]
```

Solution 2: Merge the Dual Recurrence. If we observe these two recurrence function, we can see we use `helper(root)`, we call `maxRootToAny` with left and right subtree, which is the same as `maxRootToAny(root)`. Then in `helper`, we use `helper(root.left)` to call `maxRootToAny(root.left.left)` and `maxRootToAny(root.left.right)`, which is exactly the same as `maxRootToAny(root.left)`. Thus, the above solution has one power more of complexity. It can be simplified as the

following code:

```

1 def maxRootToAny(self, root, ans):
2     if root is None:
3         return 0
4     left = self.maxRootToAny(root.left, ans)
5     right = self.maxRootToAny(root.right, ans)
6     ans[0] = max(ans[0], max(left, 0) + max(right, 0) + root
7 .val) #track the any->root->any maximum
8     # conquer: the current node
9     return root.val + max(0, max(left, right)) #track root
->any maximum
10    def maxPathSum(self, root):
11        """
12        :type root: TreeNode
13        :rtype: int
14        """
15        if root is None:
16            return 0
17        ans = [-sys.maxsize]
18        self.maxRootToAny(root, ans)
19        return ans[0]

```

The most important two lines of the code is:

```

1 ans[0] = max(ans[0], max(left, 0) + max(right, 0) + root.val
2 ) #track the any->root->any maximum
3 return root.val + max(0, max(left, right)) #track root->any
maximum

```

29.2.4 Reconstruct the Tree

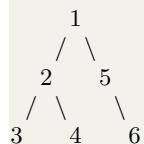
In this section, we can be asked to rearrange the node or the value of the tree either in-place or out-of-place. Unless be required to do it in-place we can always use the divide and conquer with return value and merge.

In-place Reconstruction

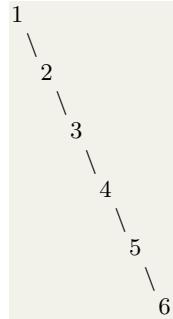
114. Flatten Binary Tree to Linked List

29.10 114. Flatten Binary Tree to Linked List (medium). Given a binary tree, flatten it to a linked list in-place.

For example, given the following tree:



The flattened tree should look like:



Solution: Inorder Traversal. For this, we first notice the flatten rule is to connect node, node.left and node.right, where node.left and node.right is already flatten by the recursive call of the function. For node 2, it will be 2->3->4. First, we need to connect node.right to node.left by setting the last node of the left's right child to be node.right.

```

1 def flatten(self, root):
2     """
3     :type root: TreeNode
4     :rtype: void Do not return anything, modify root in-
5     place instead.
6     """
7
8     if not root:
9         return
10    # preorder
11    self.flatten(root.left) # modify root.left
12    self.flatten(root.right)
13
14    # traverse the left branch to connect with the right
15    # branch
16    if root.left is not None:
17        node = root.left
18        while node.right:
19            node = node.right
20            node.right = root.right
21
22    else:
23        root.left = root.right
24    # connect node, left right
25    root.right = root.left
26    root.left = None
  
```

Out-of-place Reconstruction

1. 617. Merge Two Binary Trees
2. 226. Invert Binary Tree (Easy)

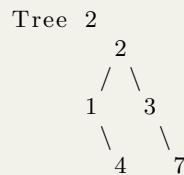
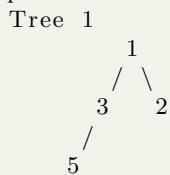
3. 654. Maximum Binary Tree(Medium)

29.11 **617. Merge Two Binary Trees.** Given two binary trees and imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not.

You need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of new tree.

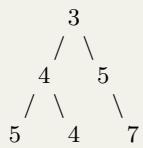
Example 1:

Input :



Output :

Merged tree :



Note: The merging process must start from the root nodes of both trees .

Solution 1: DFS+Divide and Conquer. In this problem, we just need to traverse these two trees at the same time with the same rule. When both is None which means we just reached a leaf node, we return None for its left and right child. If only one is None, then return the other according to the rule. Otherwise merge their values and assign the left subtree and right subtree to another recursive call and merge all the results to current new node.

```

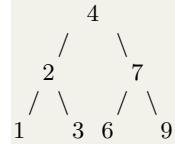
1 def mergeTrees(self, t1, t2):
2     if t1 is None and t2 is None: # both none
3         return None
4     if t1 is None and t2:
5         return t2
6     if t1 and t2 is None:
7         return t1
8     node = TreeNode(t1.val+t2.val)
9     # divide and conquer, left result and the right result
10    node.left = self.mergeTrees(t1.left, t2.left)
11    node.right = self.mergeTrees(t1.right, t2.right)
12    return node

```

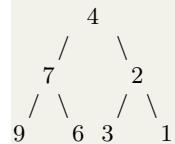
29.12 **226. Invert Binary Tree.** Invert a binary tree.

Example :

Input :



Output :



Solution 1: Divide and Conquer.

```

1 def invertTree(self, root):
2     """
3     :type root: TreeNode
4     :rtype: TreeNode
5     """
6     if root is None:
7         return None
8
9     # divide: the problem into reversing left subtree and
10    # right subtree
11    left = self.invertTree(root.left)
12    right = self.invertTree(root.right)
13    # conquer: current node
14    root.left = right
15    root.right = left
16    return root
  
```

29.13 654. Maximum Binary Tree. Given an integer array with no duplicates. A maximum tree building on this array is defined as follow:

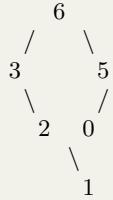
1. The root is the maximum number in the array.
2. The left subtree is the maximum tree constructed from left part subarray divided by the maximum number.
3. The right subtree is the maximum tree constructed from right part subarray divided by the maximum number.

Construct the maximum tree by the given array and output the root node of this tree.

Example 1:

Input: [3, 2, 1, 6, 0, 5]

Output: return the tree root node representing the following tree :



Note :

The size of the given array will be in the range [1, 1000].

Solution: Divide and Conquer. The description of the maximum binary tree the root, left subtree, right subtree denotes the root node is the maximum value, and the left child is the max value in the left side of the max value in the array. This fits the divide and conquer. This is so similar as the concept of **quick sort**. Which divide an array into two halves. The time complexity is $O(nlgn)$. In the worst case, the depth of the recursive tree can grow up to n, which happens in the case of a sorted nums array, giving a complexity of $O(n^2)$.

```

1 def constructMaximumBinaryTree(self, nums):
2     """
3         :type nums: List[int]
4         :rtype: TreeNode
5     """
6     if not nums:
7         return None
8     (m, i) = max((v, i) for i, v in enumerate(nums))
9     root = TreeNode(m)
10    root.left = self.constructMaximumBinaryTree(nums[:i])
11    root.right = self.constructMaximumBinaryTree(nums[i+1:])
12    return root
  
```

Monotone Queue. The key idea is:

1. We scan numbers from left to right, build the tree one node by one step;
2. We use a queue to keep some (not all) tree nodes and ensure a decreasing order;
3. For each number, we keep popping the queue until empty or a bigger number appears; 1) The kicked out smaller number is current node's left child (temporarily, this relationship may change in the future). 2) The bigger number (if exist, it will be still in stack) is

current number's parent, this node is the bigger number's right child. Then we push current number into the stack.

```

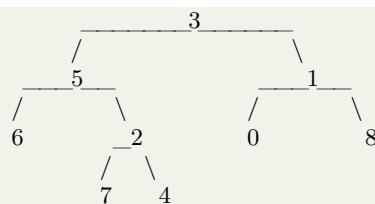
1 def constructMaximumBinaryTree(self, nums):
2     """
3     :type nums: List[int]
4     :rtype: TreeNode
5     """
6     if not nums:
7         return None
8     deQ = collections.deque()
9     for i, v in enumerate(nums):
10        node = TreeNode(v)
11        while deQ and deQ[-1].val < v:
12            node.left = deQ[-1]
13            deQ.pop()
14        if deQ:
15            deQ[-1].right = node
16        deQ.append(node)
17    return deQ[0]

```

29.2.5 Find element

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.

29.14 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, 1, 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
17            return root
18    if any([left, right]) is not None:
19        return left if left is not None else right
20    return None

```

29.2.6 Ad Hoc Problems

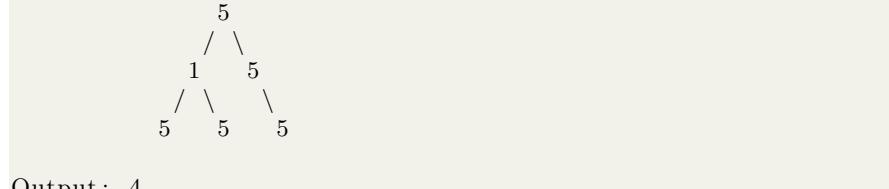
There are some other problems that are flexible and are highly customized requirements. We usually need to be more flexible with the solutions too. Sometimes, we need to write multiple functions in order to solve one problem.

1. 250. Count Unival Subtrees
2. 863. All Nodes Distance K in Binary Tree

29.15 **250. Count Unival Subtrees (medium).** Given a binary tree, count the number of uni-value subtrees. A Uni-value subtree means all nodes of the subtree have the same value.

Example :

Input: root = [5,1,5,5,5,null,5]



Output: 4

Solution 1: DFS and Divide and Conquer. First, all the leaf nodes are univalue subtree with count 1 and also it is the base case with (True, leaf.val, 1) as return. If we are at node 1, we check the left subtree and right subtree if they are univalue, and what is their value, and what is there count. Or for cases that a node only has one subtree. If the val of the subtree and the current node equals, we increase the count by one, and return (True, node.val, l_count+r_count+1). All the other cases, we only have (False, None, l_count+r_count).

```

1 def countUnivalSubtrees(self, root):
2     if not root:
3         return 0
4
5     def univalSubtree(root):
6         if root.left is None and root.right is None:
7             return (True, root.val, 1)
8         l_uni, l_val, l_count = True, None, 0
9         if root.left:
10            l_uni, l_val, l_count = univalSubtree(root.left)
11        )
12        r_uni, r_val, r_count = True, None, 0
13        if root.right:
14            r_uni, r_val, r_count = univalSubtree(root.right)
15        if l_uni and r_uni:
16            if l_val is None or r_val is None:# a node with
17                only one subtree
18                if l_val == root.val or r_val == root.val:
19                    return (True, root.val, l_count+r_count
20+1)
21                else:
22                    return (False, None, l_count+r_count)
23            if l_val == r_val == root.val:# a node with
24                both subtrees
25                return (True, root.val, l_count+r_count+1)
26            else:
27                return (False, None, l_count+r_count)
28        return (False, None, l_count+r_count)
29
30 _, _, count = univalSubtree(root)
31 return count
  
```

Or else we can use a global variable to record the subtree instead of returning the result from the tree.

```

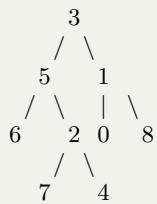
1 def countUnivalSubtrees( self , root ):
2     def helper( root ):
3         if not root: return True
4         if not root.left and not root.right:
5             self.res += 1
6             return True
7         left_res = helper( root.left )
8         right_res = helper( root.right )
9         if root.left and root.right:
10            if root.val == root.left.val and root.val ==
11                root.right.val and left_res and right_res:
12                self.res += 1
13                return True
14            return False
15        if root.left and not root.right:
16            if root.val == root.left.val and left_res:
17                self.res += 1
18                return True
19            return False
20        if root.right and not root.left:
21            if root.val == root.right.val and right_res:
22                self.res += 1
23                return True
24            return False
25    self.res = 0
26    helper( root )
27    return self.res

```

29.16 863. All Nodes Distance K in Binary Tree (medium). We are given a binary tree (with root node root), a target node, and an integer value K. (Note that the inputs "root" and "target" are actually TreeNodes.) Return a list of the values of all nodes that have a distance K from the target node. The answer can be returned in any order.

Example 1:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2



Output: [7,4,1]

Explanation:

The nodes that are a distance 2 from the target node (with value 5) have values 7, 4, and 1.

Solution 1: DFS traversal with depth to target as return.

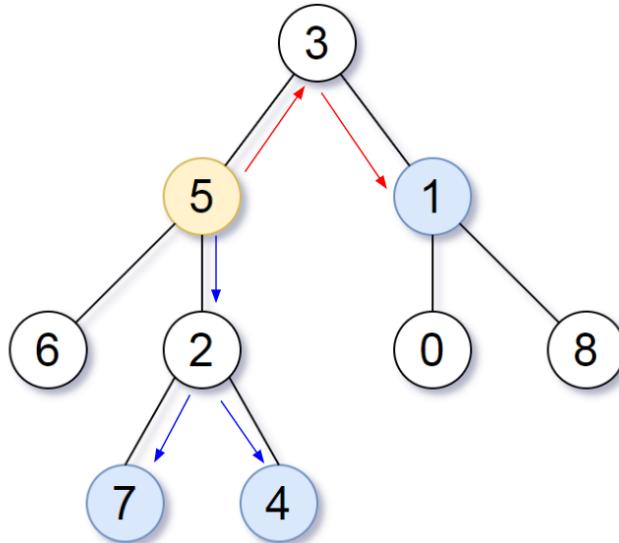


Figure 29.1: Two Cases of K Distance Nodes marked in blue and red arrows.

There are different cases with path that has target as denoted in Fig 29.1: 1. target is the starting point, we traverse the target downwards to get nodes that is K distance away from target. 2. target is the ending point, we need to traverse back to its parents, and first check the distance of the parent node with the target to see if it is K, and second we use another function to find K-distance away nodes on the other branch of the parent node. Because we do not have pointer back to its parents directly, we use recursive tree traversal so that we can return to the parent node with its distance to the target. Therefore, we need two helper functions. The first function *getDistanceK* takes a starting node, and a distance K, to return a list of K distance downwards from starting point. The second function *getDepth* is designed to do the above task, when we find the target in the tree traversal, we return 0, for empty node return -1.

```

1 def distanceK(self, root, target, K):
2     if not root:
3         return []
4     def getDistanceK(target, K):
5         ans = []
6         # from target to K distance
7         q = [target]
8         d = 0
9         while q:
10             if d == K:
11                 ans += [n.val for n in q]
12                 break
13             nq = []
  
```

```

14     for n in q:
15         if n.left:
16             nq.append(n.left)
17         if n.right:
18             nq.append(n.right)
19         q = nq
20         d += 1
21     return ans
22
23 # get depth of target
24 def getDepth(root, target, K, ans):
25     if not root:
26         return -1
27     if root == target:
28         return 0
29     # conquer
30     left = getDepth(root.left, target, K, ans)
31     right = getDepth(root.right, target, K, ans)
32     if left == -1 and right == -1:
33         return -1
34     else:
35         dis = 0
36         if left != -1:
37             dis = left+1
38             if root.right:
39                 ans += getDistanceK(root.right, K-dis
40 -1)
41         else:
42             dis = right + 1
43             if root.left:
44                 ans += getDistanceK(root.left, K-dis-1)
45             if dis == K:
46                 ans.append(root.val)
47             return dis
48
49     ans = getDistanceK(target, K)
50     getDepth(root, target, K, ans)
51     return ans

```

Solution 2: DFS to annotate parent node + BFS to K distance nodes. In solution 1, we have two cases because we can't traverse to its parents node directly. If we can add the parent node to each node, and the whole tree would become a acyclic direct graph, thus, we can use BFS to find all the nodes that are K distance away. This still has the same complexity.

```

1 def distanceK(self, root, target, K):
2     if not root:
3         return []
4     def dfs(node, par = None):
5         if node is None:
6             return
7         node.par = par

```

```

8         dfs(node.left, node)
9         dfs(node.right, node)
10    dfs(root)
11    seen = set([target])
12    q = [target]
13    d = 0
14    while q:
15        if d == K:
16            return [node.val for node in q]
17        nq = []
18        for n in q:
19            for nei in [n.left, n.right, n.par]:
20                if nei and nei not in seen:
21                    seen.add(nei)
22                    nq.append(nei)
23        q = nq
24        d += 1
25    return []

```

29.3 Binary Search Tree (BST)

29.3.1 BST Rules

1. 98. Validate Binary Search Tree (Medium)
2. 99. Recover Binary Search Tree(hard)
3. 426. Convert Binary Search Tree to Sorted Doubly Linked List (medium)

29.17 **98. Validate Binary Search Tree (medium)** Given a binary tree, determine if it is a valid binary search tree (BST). Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

```

1 Example 1:
2
3 Input:
4      2
5      / \
6      1   3
7 Output: true
8
9 Example 2:
10

```

```

11      5
12      / \
13      1   4
14      / \
15      3   6
16 Output: false
17 Explanation: The input is: [5,1,4,null,null,3,6]. The root
               node's value
18                   is 5 but its right child's value is 4.

```

Solution1: Limit the value range for subtrees: top-down. We start from the root, which should be in range $[-\infty, +\infty]$. And the left subtree should be limited into $[-\infty, \text{root.val}]$, and right in $[\text{root.val}, +\infty]$. The Code is simple and clean:

```

1 def isValidBST(self, root, minv=float("-inf"), maxv=float("inf")):
2     """
3         :type root: TreeNode
4         :rtype: bool
5     """
6     if root is None:
7         return True
8
9     if (minv < root.val < maxv):
10        return self.isValidBST(root.left, minv, root.val)
11    and self.isValidBST(root.right, root.val, maxv)
12    return False

```

Solution 2: Limit the value range for parent node: bottom-up. We traverse the tree, and we return values from the None node, then we have three cases:

```

1) both subtrees are None # a leaf
2       return (True, root.val, root.val)
3) both subtrees are not None: # a subtree with two
   branches
4       check if l2 < root.val < r1:
5       merge the range to:
6       return (True, l1, r2)
7) one subtree is None: # a subtree with one branches:
8       only check one of l2, r1 and merge accordingly

```

Solution 2: Using inorder. If we use inorder, then the tree resulting list we obtained should be strictly increasing.

```

1 def isValidBST(self, root):
2     if root is None:
3         return True
4
5     def inOrder(root):
6         if not root:
7             return []

```

```

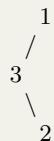
8     return inOrder(root.left) + [root.val] + inOrder(
9         root.right)
10    ans = inOrder(root)
11    pre = float("-inf")
12    for v in ans:
13        if v <= pre:
14            return False
15    pre = v
16    return True

```

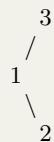
- 29.18 99. Recover Binary Search Tree (hard).** Two elements of a binary search tree (BST) are swapped by mistake. Recover the tree without changing its structure.

Example 1:

Input: [1,3,null,null,2]



Output: [3,1,null,null,2]



Example 2:

Input: [3,1,4,null,null,2]



Output: [2,1,4,null,null,3]



Follow up: A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

Solution 1: Recursive InOrder Traversal and Sorting, O(n) space. The same as validating a BST, the inorder traversal of a valid BST must have a sorted order. Therefore, we obtain the inorder traversed list, and sort them by the node value, and compared the sorted list and the unsorted list to find the swapped nodes.

```

1 def recoverTree(self, root):
2     """
3         :type root: TreeNode
4         :rtype: void Do not return anything, modify root in-
5             place instead.
6     """
7     def inorder(root):
8         if not root:
9             return []
10        return inorder(root.left) + [root] + inorder(root.
11        right)
12
13    ans = inorder(root)
14    sans = sorted(ans, key = lambda x: x.val)
15    # swap
16    for x, y in zip(ans, sans):
17        if x != y:
18            x.val, y.val = y.val, x.val
            break

```

Solution 2: Iterative Traversal: O(1) space. The inorder traversal for each example are:

Example 1: [3, 2, 1], need to switch 3, 1
 Example 2: [1, 3, 2, 4], need to switch 3, 2

If we observe the inorder list: if we check the previous and current pair, if it is dropping as (3,2), (2,1), then we call this dropping pairs. In example 2, there is only one pair (3,2). This is the two possible cases when we swap a pair of elements in a sorted list. If we use the inorder iterative traversal, and record the pre, cur dropping pairs, then it is straightforward to do the swapping of the dropping pair or just one pair.

```

1 def recoverTree(self, root):
2     cur, pre, stack = root, TreeNode(float("-inf")), []
3     drops = []
4     # inorder iterative: left root, right
5     while stack or cur:
6         while cur:
7             stack.append(cur)
8             cur = cur.left
9             cur = stack.pop()
10            if cur.val < pre.val:
11                drops.append((pre, cur))
12                pre, cur = cur, cur.right

```

```

13
14     drops[0][0].val, drops[-1][1].val = drops[-1][1].val,
      drops[0][0].val

```

- 29.19 426. Convert Binary Search Tree to Sorted Doubly Linked List (medium)** Convert a BST to a sorted circular doubly-linked list in-place. Think of the left and right pointers as synonymous to the previous and next pointers in a doubly-linked list. One example is shown in Fig. 29.2.

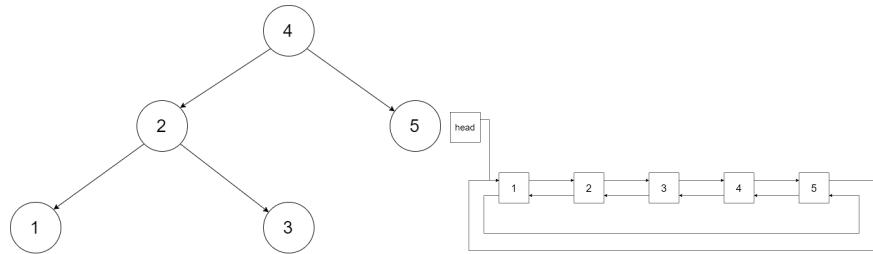


Figure 29.2: Example of BST to DLL

Analysis As we observe the example, for each node in the doubly linked list (dll), its predecessor and successor is the same as the same node in BST. As we have learned the concept of predecessor and successor in Chapter ??, we know how to find the predecessor and successor individually for each node. However, in this scene, it would be more useful with the inorder traversal, wherein we can use divide and conquer to obtain the left sorted list and the right sorted list for each node. More than this, we need to make the dll, we have two choices to do this: 1) Use our learned inorder traversal to generate a list, and then generate the dll from the list of BST nodes. 2) Combine the inorder traversal together with the linking process.

Solution 1: Inorder traversal + Doubly linked List Connect. This process is straightforward, we need to handle the case where the BST only has one node, or for BST that has at least two nodes. For the second case, we should handle the head and tail node separately due to its different linking rule:

```

1 def treeToDoublyList(self, root):
2     """
3         :type root: Node
4         :rtype: Node
5     """
6     if not root:
7         return None
8
9     def treeTraversal(root):
10         if not root:

```

```

11     return []
12     left = treeTraversal(root.left)
13
14     right = treeTraversal(root.right)
15     return left + [root] + right
16
17     sortList = treeTraversal(root)
18     if len(sortList) == 1:
19         sortList[0].left = sortList[0]
20         sortList[0].right = sortList[0]
21     return sortList[0]
22
23     for idx, node in enumerate(sortList):
24         if idx == 0:
25             node.right = sortList[idx+1]
26             node.left = sortList[-1]
27         elif idx == len(sortList) - 1:
28             node.right = sortList[0]
29             node.left = sortList[idx-1]
30         else:
31             node.right = sortList[idx+1]
32             node.left = sortList[idx-1]
33     return sortList[0]

```

Solution 2: Inorder traversal together with linking process.

We use divide and conquer method and assuming the left and right function call gives us the head of the dll on each side. With left_head and right_head, we just need to link these two separate dlls with current node in the process of inorder traversal. The key here is to find the tail left dll, and link them like: left_tail+current_node+right_head, and link left_head with right_tail. With dlls, to find the tail from the head, we just need to use head.left.

```

1 def treeToDoublyList(self, root):
2     """
3     :type root: Node
4     :rtype: Node
5     """
6     if not root: return None
7
8     left_head = self.treeToDoublyList(root.left)
9     right_head = self.treeToDoublyList(root.right)
10    return self.concat(left_head, root, right_head)
11
12    """
13    Concatenate a doubly linked list (prev_head), a node
14    (curr_node) and a doubly linked list (next_head) into
15    a new doubly linked list.
16    """
17    def concat(self, left_head, curr_node, right_head):
18        # for current node, it has only one node, head and tail
19        # is the same

```

```

20     new_head, new_tail = curr_node, curr_node
21
22     if left_head:
23         # find left tail
24         left_tail = left_head.left
25         # connect tail with current node
26         left_tail.right = curr_node
27         curr_node.left = left_tail
28         # new_head points to left_head
29         new_head = left_head
30
31     if right_head:
32         right_tail = right_head.left
33         # connect head with current node
34         curr_node.right = right_head
35         right_head.left = curr_node
36         new_tail = right_tail # new_tail points to
37         right_tail
38
39     new_head.left = new_tail
40     new_tail.right = new_head
41     return new_head

```

29.3.2 Operations

In this section, we should problems related to operations we introduced in section ??, which include SEARCH, INSERT, GENERATE, DELETE. LeetCode Problems include:

1. 108. Convert Sorted Array to Binary Search Tree
2. 96. Unique Binary Search Trees

29.20 108. Convert Sorted Array to Binary Search Tree. Given an array where elements are sorted in ascending order, convert it to a height balanced BST. For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

Example :

Given the sorted array: `[-10, -3, 0, 5, 9]`,

One possible answer is: `[0, -3, 9, -10, null, 5]`, which represents the following height balanced BST:

```
\begin{lstlisting}
      0
     / \
    -3   9
   /   /
  -10  5
\end{lstlisting}
```

Solution: Binary Searching. use the binary search algorithm, the stop condition is when the $l > r$.

```

1 def sortedArrayToBST( self ,  nums ) :
2     """
3         :type  nums:  List [ int ]
4         :rtype:  TreeNode
5     """
6     def generatebalancedBST( l , r ) :
7         if  l > r :
8             return None
9         m = ( l + r ) // 2
10        tree = TreeNode( nums [ m ] )
11        tree . left = generatebalancedBST( l , m - 1 )
12        tree . right = generatebalancedBST( m + 1 , r )
13        return tree
14    return generatebalancedBST( 0 , len ( nums ) - 1 )

```

109. Convert Sorted List to Binary Search Tree, the difference is here we have a linked list, we can convert the linked list into a list nums

29.21 96. Unique Binary Search Trees

Given n , how many structurally unique BST's (binary search trees) that store values $1 \dots n$?

For example ,

Given $n = 3$, there are a total of 5 unique BST's .

```

      1           3           3           2           1
       \         /           /           / \         \
        3         2           1           1   3       2
         /         /           \         /   \       \
        2         1           2           1     3

```

Solution: When we read the signal, list all of it, we need to use for loop, to pose each element as root, and the left side is left tree, the right side is used for the right tree. Use DPS: We generated all the BST that use ith node as root

```

1 def numTrees( self ,  n ) :
2     """
3         :type  n:  int
4         :rtype:  int
5     """
6     def constructAllBST( start , end ) :
7         if  start > end :
8             return [ None ]
9
10        #go through the start to end , and use the ith
11        #as root
12        rslt = []
13        leftsubs , rightsubs = [ ] , [ ]
14        for i in xrange( start , end + 1 ) :

```

```

14
15     leftsubs=constructAllBST(start,i-1)
16     rightsubs=constructAllBST(i+1,end)
17     for leftnode in leftsubs:
18         for rightnode in rightsubs:
19             node = TreeNode(i)
20             node.left=leftnode
21             node.right=rightnode
22             rslt.append(node)
23     return rslt
24
25 rslt= constructAllBST(1,n)
26     return len(rslt)
27

```

If we only need length, a slightly better solution showing as follows.

```

1 def numTrees(self, n):
2     """
3     :type n: int
4     :rtype: int
5     """
6     def constructAllBST(start, end):
7         if start > end:
8             return 1
9
10        #go through the start to end, and use the ith
11        #as root
12        count = 0
13        leftsubs, rightsubs = [], []
14        for i in xrange(start, end+1):
15            leftsubs = constructAllBST(start, i-1)
16            rightsubs = constructAllBST(i+1, end)
17            count += leftsubs * rightsubs
18        return count
19
20 rslt= constructAllBST(1,n)
21     return rslt
22

```

However, it still cant pass the test, try the bottom up iterative solution with memorization: $T(start, end) = T(start, i - 1) * T(i + 1, end)$. How to explain this?

```

1 def numTrees1(self, n):
2     res = [0] * (n+1)
3     res[0] = 1
4     for i in xrange(1, n+1): #when i=2, j=[0,1] res[2] =
5         res[0]*res[2-1-0] + res[1]*res[2-1-1]
6         for j in xrange(i): #i [1,n], j =[0,i), the case if
7             for one node,
8                 res[i] += res[j] * res[i-1-j]
9     return res[n]

```

Using math:

```

1 # Catalan Number (2n)!/((n+1)!*n!)
2 def numTrees(self, n):
3     return math.factorial(2*n)/(math.factorial(n)*math.
factorial(n+1))

```

29.3.3 Find certain element of the tree

successor or predecessor:285. Inorder Successor in BST, 235. Lowest Common Ancestor of a Binary Search Tree

1. 285. Inorder Successor in BST
2. 235. Lowest Common Ancestor of a Binary Search Tree
3. 230. Kth Smallest Element in a BST
4. 270. Closest Binary Search Tree Value
5. 272. Closest Binary Search Tree Value II
6. 426. Convert Binary Search Tree to Sorted Doubly Linked List (find the predecessor and successor)

Lowest Common Ancestor(LCA) The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself)." e.g., if u=5,w=19, then we first node when we recursively visiting the tree that is within [u,w], then the LCA is 14. Compared with LCA for binary tree, because of the searching property of searching tree, it is even simpler:

```

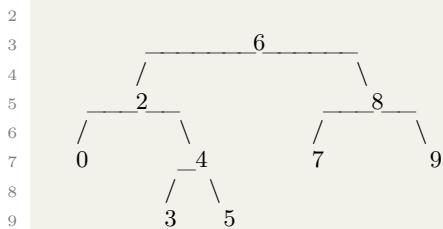
1 traverse the tree:
2 if node.val is in [s, b], return node is LCA
3 if node.val > b, traverse node.left
4 if node.val < s, traverse node.right
5

```

235. Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

```
1 Given binary search tree: root = [6,2,8,0,4,7,9,null,null,3,5]
```



```

10
11 Example 1:
12
13 Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
14 Output: 6
15 Explanation: The LCA of nodes 2 and 8 is 6.
16
17 Example 2:
18
19 Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
20 Output: 2
21 Explanation: The LCA of nodes 2 and 4 is 2, since a node can be
   a descendant of itself
22           according to the LCA definition.

```

```

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     s = min(p.val, q.val)
9     b = max(p.val, q.val)
10    def LCA(node):
11        if not node:
12            return None
13        if node.val > b:
14            return LCA(node.left)
15        if node.val < s:
16            return LCA(node.right)
17        # current node [s, b], then this is LCA
18        return node
19
20    return LCA(root)

```

In order traverse can be used to sorting. e.g. 230. Kth Smallest Element in a BST

Successor 285. Inorder Successor in BST

First, we can follow the definition, use the inorder traverse to get a list of the whole nodes, and we search for the node p and return its next in the lst.

```

1 #takes 236 ms
2 def inorderSuccessor(self, root, p):
3     """
4         :type root: TreeNode
5         :type p: TreeNode
6         :rtype: TreeNode
7     """
8     lst = []
9     def inorderTravel(node):

```

```

10     if not node:
11         return None
12     inorderTravel(node.left)
13     lst.append(node)
14     inorderTravel(node.right)
15
16     inorderTravel(root)
17
18     for i in xrange(len(lst)):
19         if lst[i].val==p.val:
20             if i+1<len(lst):
21                 return lst[i+1]
22             else:
23                 return None

```

However, the definition takes $O(N+N)$, also space complexity is $O(N)$. A slightly better version is when we traverse to find this node, the successor is the last bigger value. So we keep recording the succ

```

1 #takes 108ms
2 def inorderSuccessor(self, root, p):
3     """
4         :type root: TreeNode
5         :type p: TreeNode
6         :rtype: TreeNode
7     """
8     #find min(p.right)
9     succ = None
10    while root:
11        if p.val < root.val: #node is bigger than p, go to
12            left
13                succ = root #it might be the successor since its
14            bigger
15                root = root.left
16            else: #node has the same value or smaller value than
17                p, go to right
18                root = root.right
19    return succ

```

235. Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).”

For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

Solution: we traverse the tree to find the first element that is in the range.

```

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     s=min(p.val,q.val)
9     b=max(p.val,q.val)
10    def LCA(node):
11        if not node:
12            return None
13        if node.val>b:
14            return LCA(node.left)
15        if node.val<s:
16            return LCA(node.right)
17        return node
18
19    return LCA(root)

```

230. Kth Smallest Element in a BST

Solution: Here, DFS-inorder-recursive; if we use python, then we need to use list as global variable

```

1 def kthSmallest(self , root , k):
2     """
3         :type root: TreeNode
4         :type k: int
5         :rtype: int
6     """
7     count = [k] #use list to function as global variable
8     num=[-1]
9     #inorder traversal
10    def inorder(node):
11        if node.left:
12            inorder(node.left)
13            count[0]-=1 #visit
14        if count[0]==0:
15            num[0]=node.val
16            return
17        if node.right:
18            inorder(node.right)
19    inorder(root)
20    return num[0]

```

Second: DFS iterative

```

1 def kthSmallest(self , root , k):
2     """
3         :type root: TreeNode
4         :type k: int
5         :rtype: int
6     """
7     count = k
8     num=-1

```

```

9      #DFS iterative
10     #1: add all the left nodes
11     while root:
12         stack.append(root)
13         root=root.left
14
15     #visit stack, if node.right exist, then add in the stack
16     while stack:
17         node=stack.pop()
18         count-=1
19         if count==0:
20             return node.val
21         #check if the right exist
22         right=node.right
23         while right:
24             stack.append(right)
25             right=right.left #keep getting the successor

```

270. Closest Binary Search Tree Value

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note:

Given target value is a floating point.

You are guaranteed to have only one unique value in the BST that is closest to the target.

```

1 from sys import maxint
2 class Solution:
3     def closestValue(self, root, target):
4         """
5             :type root: TreeNode
6             :type target: float
7             :rtype: int
8         """
9         mind= maxint #current smallest
10        rslt= -1
11        while root:
12            if root.val>target:
13                if root.val-target<mind:
14                    mind = root.val-target
15                    rslt=root.val
16                    root=root.left
17            elif root.val<target:
18                if target-root.val<mind:
19                    mind = target - root.val
20                    rslt=root.val
21                    root=root.right
22            else:
23                return root.val
24        return rslt

```

272. Closest Binary Search Tree Value II

29.3.4 Trim the Tree

maybe with value in the range: 669. Trim a Binary Search Tree

669. Trim a Binary Search Tree

Given a binary search tree and the lowest and highest boundaries as L and R, trim the tree so that all its elements lies in [L, R] ($R \geq L$). You might need to change the root of the tree, so the result should return the new root of the trimmed binary search tree.

Example 2:

```

1 Input :
2      3
3      / \
4      0   4
5      \
6      2
7      /
8      1
9
10     L = 1
11     R = 3
12
13 Output :
14      3
15      /
16      2
17      /
18      1
19

```

Solution: Based on F1, if the value of current node is smaller than L, suppose at 0, then we delete its left child, `node.left = None`, then we check its right size, go to `node.right`, we return `node = goto(node.right)`, if it is within range, then we keep checking left, right, and return current node

```

1 def trimBST(self, root, L, R):
2     """
3         :type root: TreeNode
4         :type L: int
5         :type R: int
6         :rtype: TreeNode
7     """
8     def trimUtil(node):
9         if not node:
10             return None
11         if node.val < L:
12             node.left=None #cut the left , the left subtree
13             is definitely smaller than L
14             node = trimUtil(node.right) #node = node.right #
15             check the right
16             return node
17         elif node.val > R:
18             node.right=None

```

```

17         node=trimUtil(node.left)
18     return node
19 else:
20     node.left=trimUtil(node.left)
21     node.right=trimUtil(node.right)
22     return node
23 return trimUtil(root)

```

A mutant of this is to split the BST into two, one is smaller or equal to the given value, the other is bigger.

29.3.5 Split the Tree

Split the tree

with a certain value ,776. Split BST

776. Split BST

Given a Binary Search Tree (BST) with root node root, and a target value V, split the tree into two subtrees where one subtree has nodes that are all smaller or equal to the target value, while the other subtree has all nodes that are greater than the target value. It's not necessarily the case that the tree contains a node with value V.

Additionally, most of the structure of the original tree should remain. Formally, for any child C with parent P in the original tree, if they are both in the same subtree after the split, then node C should still have the parent P.

You should output the root TreeNode of both subtrees after splitting, in any order.

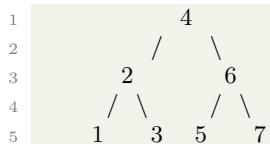
Example 1:

```

1 Input: root = [4,2,6,1,3,5,7], V = 2
2 Output: [[2,1],[4,3,6,null,null,5,7]]
3 Explanation:
4 Note that root, output[0], and output[1] are TreeNode objects,
   not arrays.

```

The given tree [4,2,6,1,3,5,7] is represented by the following diagram:



Solution: The coding is quite similar as the trimming.

```

1 class Solution(object):
2     def splitBST(self, root, V):
3         """
4             :type root: TreeNode
5             :type V: int
6             :rtype: List[TreeNode]
7         """

```

```

8     def splitUtil(node):
9         if not node:
10            return (None, None)
11        if node.val <= V:
12            sb1, sb2 = splitUtil(node.right) #the left
13            subtree will satisfy the condition, split the right subtree
14            node.right=sb1 #Now set the right subtree with
15            sb1 that
16            return (node, sb2)
17        else:
18            sb1, sb2=splitUtil(node.left) #the right subtree
19            satisfy the condition, split the left subtree
            node.left=sb2
            return (sb1, node)
        return list(splitUtil(root))

```

29.4 Exercise

29.4.1 Depth

104. Maximum Depth of Binary Tree (Easy)

Given a binary tree, find its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Note: A leaf is a node with no children.

```

1 Example:
2
3 Given binary tree [3,9,20,null,null,15,7],
4
5      3
6      / \
7      9   20
8      /   \
9      15   7
10
11 return its depth = 3.

```

DFS+Divide and conquer.

```

1 def maxDepth(self, root):
2     if not root:
3         return 0
4     if not root.left and not root.right:
5         return 1
6     depth = -sys.maxsize
7     if root.left:
8         depth = max(depth, self.maxDepth(root.left))
9     if root.right:
10        depth = max(depth, self.maxDepth(root.right))
11    return depth+1

```

559. Maximum Depth of N-ary Tree (Easy)

Given a n-ary tree, find its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

```

1 # Definition for a Node.
2 class Node(object):
3     def __init__(self, val, children):
4         self.val = val
5         self.children = children
6
7     def maxDepth(self, root):
8         if not root:
9             return 0
10        children = root.children
11        if not any(children): # a leaf
12            return 1
13        depth = -sys.maxsize
14        for c in children:
15            if c:
16                depth = max(depth, self.maxDepth(c))
17        return depth+1

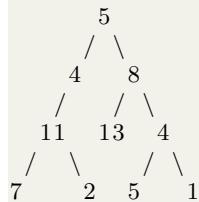
```

29.4.2 Path

113. Path Sum II (medium). Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum. *Note: A leaf is a node with no children.*

Example:

Given the below binary tree and sum = 22,



Return:

```
[
    [5,4,11,2],
    [5,8,4,5]
]
```

```

1 def pathSumHelper(self, root, sum, curr, ans):
2     if root is None: # this is for one branch tree
3         return
4     if root.left is None and root.right is None: # a leaf as
5         base case
6             if sum == root.val:

```

```

6         ans.append(curr+[root.val])
7     return
8
9     self.pathSumHelper(root.left, sum-root.val, curr+[root.val],
10    ans)
11
12    self.pathSumHelper(root.right, sum-root.val, curr+[root.val]
13    ], ans)
14
15 def pathSum(self, root, sum):
16     """
17     :type root: TreeNode
18     :type sum: int
19     :rtype: List[List[int]]
20     """
21
22     ans = []
23     self.pathSumhelper(root, sum, [], ans)
24     return ans

```

257. Binary Tree Paths

Given a binary tree, return all root-to-leaf paths.

Note: A leaf is a node with no children.

```

1 Example:
2 Input:
3
4      1
5      /   \
6     2     3
7     \   /
8     5   2
9 Output: ["1->2->5", "1->3"]
10 Explanation: All root-to-leaf paths are: 1->2->5, 1->3

```

Root to Leaf. Be careful that we only collect result at the leaf, and for the right tree and left tree we need to make sure it is not None:

```

1 def binaryTreePaths(self, root):
2     """
3     :type root: TreeNode
4     :rtype: List[str]
5     """
6
7     def dfs(root, curr, ans):
8         if root.left is None and root.right is None: # a leaf
9             ans.append(curr+str(root.val))
10            return
11        if root.left:
12            dfs(root.left, curr+str(root.val)+"->", ans)
13        if root.right:
14            dfs(root.right, curr+str(root.val)+"->", ans)
15        if root is None:
16            return []
17        ans = []
18        dfs(root, "", ans)
19        return ans

```

543. Diameter of Binary Tree

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.

```

1 Example :
2 Given a binary tree
3
4      1
5      / \
6      2   3
7      / \
8      4   5
9
10 Return 3, which is the length of the path [4,2,1,3] or
     [5,2,1,3].

```

Root to Any with Global Variable to track the any to any through root.

```

1 def diameterOfBinaryTree(self, root):
2     """
3     :type root: TreeNode
4     :rtype: int
5     """
6     # this is the longest path from any to any
7
8     def rootToAny(root, ans):
9         if not root:
10             return 0
11         left = rootToAny(root.left, ans)
12         right = rootToAny(root.right, ans)
13         ans[0] = max(ans[0], left+right) # track the any to any
14         return max(left, right) + 1 #get the maximum depth of
15         root to any
16     ans = [0]
17     rootToAny(root, ans)
18     return ans[0]

```


30

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 ??.

30.1 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.

30.1.1 Explicit BFS/DFS

30.1.2 Implicit BFS/DFS

30.1 582. 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.

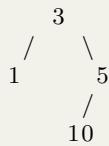
We use two list of integers to represent a list of processes, where the first list contains PID for each process and the second list contains the corresponding PPID.

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)

```

30.2 Connected Components

30.2 130. 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:

```

```

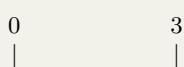
8         return
9     moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]
10    # find all connected components to the edge 0, and mark
11    # them as -1,
12    # then flip all 0s in the other parts
13    # change the -1 to 0s
14    def dfs(x, y): # (x, y) is the edge 0s
15        for dx, dy in moves:
16            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
34    # # first and last row
35    for j in range(cols):
36        if board[0][j] == 'O':
37            board[0][j] = '-1'
38            dfs(0, j)
39        if board[rows-1][j] == 'O':
40            board[rows-1][j] = '-1'
41            dfs(rows-1, j)
42
43    for i in range(rows):
44        for j in range(cols):
45            if board[i][j] == 'O':
46                board[i][j] = 'X'
47            elif board[i][j] == '-1':
48                board[i][j] = 'O'
49            else:
50                pass

```

30.3 323. 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
```

30.3 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 acutally 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

30.4 200. 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 visted 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

```

30.5 934. 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
```

30.4 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.

30.6 943. 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: "gctaaggatcatgcata"
```

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)] # key is the index, value (index: length of suffix with
12         # the next prefix)
13         if not A:
14             return G
15         for i, s in enumerate(A):
16             for j in range(n):
17                 if i == j:
18                     continue
19
20                 t = A[j]
21                 m = min(len(s), len(t))
22                 for l in range(m, 0, -1): #[n, 1]
23                     if s[-l:] == t[0:l]: #[suffix and
prefix
24                                     G[i][j] = l

```

```

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.

Part XI

Appendix

31

Python Knowledge Base

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.

31.1 Python Overview

In this section, we provide a well-organized overview of how Python works as an **object-oriented programming language** in Subsection 31.1.1, what is the components of Python: built-in Data types, built-in modules, third party packages/libraries, frameworks (Subsection 31.1.2). And in this book, we selectively introduce the most useful ones that considered for the purpose of learning algorithms and passing coding interviews.

31.1.1 Understanding Object

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' with its identifier (variable names a, b, c in the code). The identifier of the object acts as a pointer to the object's location in memory. The built-in function **id()** returns the identity of an object as an integer which usually corresponds to the object's location in memory. **is** operator can be used 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. We give a code snippet:

```

1 # mutable behavior: pointer or the identity will never be
2     changed
3 a = [1, 2, 3]
4 b = [1, 2, 3]
5 c = a
6 print(a == b, a is b, type(a) is type(b))
7 print(a == c, a is c, type(a) is type(c))
8 print('id a:%d, b:%d, c:%d'%( id(a), id(b), id(c)))
9 a[2] = 4
10 a += [5]
11 print('id after change, a:%d, c:%d, a:%s, c%s' % ( id(a), id(c),
12     a, c))
13 # output
14 '''
15 True False True
16 True True True
17 id a:2213525851016, b:2213525898248, c:2213525851016
18 id after change, a:2213525851016, c:2213525851016, a:[1, 2, 4,
19     5], c[1, 2, 4, 5]
20 '''

```

Mutable vs Immutable And all objects can be either mutable or immutable. A mutable object can change its state or contents and immutable objects can not but rather return new objects when attempting to update. For immutable object, when an instance is firstly created, such as 'abc', it is

saved at a certain memory location. When subsequent reassignment of this instance to another identifier. The resulting of different mutability can be reflected on the identity of identifiers. In the following code snippet, identifiers are all pointers pointing to the same physical address in the memory, which explains why at first a, b, c are all sharing the same identity. When we change the value of the variable a, it is actually pointed to another memory address which saves the copied value of original 'abc' together with 'd', we change a's identity too.

```

1 # immutable behavior
2 a = 'abc'
3 b = 'abc'
4 c = a
5 # for immutable data type: if two objects have the same value,
6 # they are just two pointers to that value in the memory,
7 # thus they have same id
8 print(a == b, a is b, type(a) is type(b))
9 print(b == c, b is c, type(b) is type(c))
10
11 # immutable behavior: pointer or identity will be changed if we
12 # want to change its value
12 print('id before change', id(a), id(b), id(c))
13 a = a + 'd'
14 print('id after change', id(a), id(b), id(c))
15 # output
16 '''
17 True True True
18 True True True
19 id before change 2213270621520 2213270621520 2213270621520
20 id after change 2213272148320 2213270621520 2213270621520
21 '''

```

31.1.2 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 III in Python in Section 31.9 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/      # first subpackage
        __init__.py
        info.py
        points.py
        transactions.py

    create/   # second subpackage
        __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 artifical intelligence filed, 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.

31.2 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 6), and two special operators like the identity operator or the membership operator.

31.2.1 Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Table 31.1: 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

31.2.2 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`.

31.2.3 Comparison Operators

Comparison operators are used to compare values. It either returns True or False according to the condition.

Table 31.2: Comparison operators in Python

Operator	Description	Example
<code>></code>	Greater than - True if left operand is greater than the right	<code>x > y</code>
<code><</code>	Less than - True if left operand is less than the right	<code>x < y</code>
<code>==</code>	Equal to - True if both operands are equal	<code>x == y</code>
<code>!=</code>	Not equal to - True if operands are not equal	<code>x != y</code>
<code>>=</code>	Greater than or equal to - True if left operand is greater than or equal to the right	<code>x >= y</code>
<code><=</code>	Less than or equal to - True if left operand is less than or equal to the right	<code>x <= y</code>

31.2.4 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
- 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 31.3: 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

31.2.5 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 31.4: 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 31.5: 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

31.3 Function

31.3.1 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.

31.3.2 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
```

31.3.3 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

```

31.4 Class

31.4.1 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, $x.\underline{\underline{add}}(y)$, and an indexing operation, $x[k]$, is mapped to $x.\underline{\underline{getitem}}(k)$. 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 31.6. A good and useful way to imple-

Table 31.6: 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>__nonzero__(self)</code>	Returns 0 or 1 for truth-value testing
<code>__unicode__(self)</code>	Creates a Unicode string representation

ment 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)

```

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 31.7 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 31.7: 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>

31.4.2 Class Syntax

31.4.3 Inheritance

31.4.4 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()`.

31.5 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

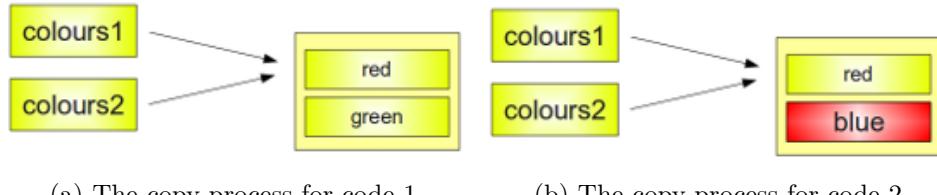


Figure 31.1: Copy process

visualize this process as follows: However, we often need to do copy and leave the original list or string unchanged. Because there are a variety of list, from one dimensional, two-dimensional to multi-dimensional.

31.5.1 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:

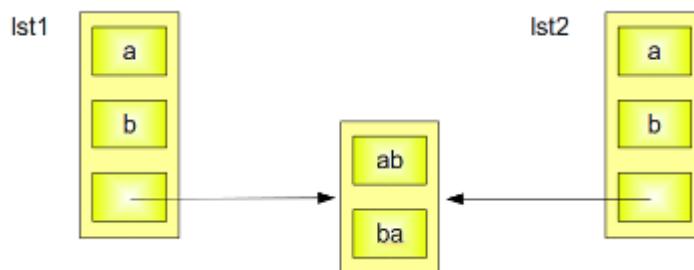


Figure 31.2: Caption

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.

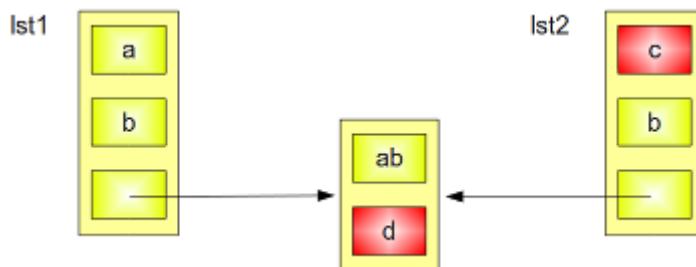


Figure 31.3: Caption

31.5.2 Iterables, Generators, and Yield

<https://pythontips.com/2013/09/29/the-python-yield-keyword-explained/>. Seems like it can not yield a list.

31.5.3 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.

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' ] ]

```

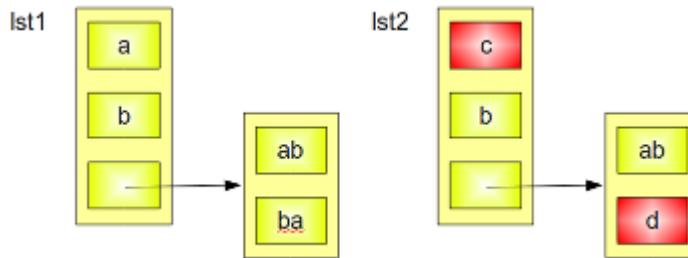


Figure 31.4: Caption

This section is cited from Need to modify.

31.6 Global Vs nonlocal

31.7 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)

```

31.8 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)
5

```

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.

31.9 Supplemental Python Tools

31.9.1 Re

31.9.2 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

```

31.9.3 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 31.8. Most of them we have learned in Part III, 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 31.8: 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.