

The Comprehensive Coding Interview Guide

Learning Data Structures and Algorithms
with LeetCode

LI YIN¹

February 17, 2019

¹<https://liyinscience.com>

Contents

0 Preface	1
I Introduction	7
1 Coding Interview and Resources	9
1.1 Coding Interviews	9
1.1.1 Process of Interviews	9
1.1.2 The Role of Coding Interviews	9
1.2 Tips and Resources	9
1.2.1 Tips to Preparation	10
1.2.2 Resources	10
1.2.3 5 Tips to the Process	11
1.3 LeetCode	12
1.3.1 Special Features	12
2 Data Structures and Algorithms	17
2.1 Four Problem Solving Paradigms	18
2.2 Problem Modeling	20
2.3 Complexity Analysis	21
2.3.1 Asymptotic Notations	21
2.3.2 Three Cases	22
2.3.3 Time and Space Complexity	22
2.3.4 Big-O Cheat Sheet	23
II Fundamental Algorithm Design and Analysis	27
3 Iteration and Recursion	31

3.1	Iteration VS Recursion	31
3.2	Recursion	32
3.3	Summary	33
4	Divide and Conquer	35
4.1	Dividing and Recurrence Function	36
4.2	Divide and Conquer	38
4.3	More Examples	39
5	Algorithm Analysis	43
5.1	Time Complexity	43
5.1.1	Solve Recurrence Function	44
5.1.2	Time Complexity	44
5.2	Space Complexity	46
5.2.1	Summary	46
5.2.2	More Examples	46
5.3	Amortized Analysis	47
III	Footstone: Data Structures	49
6	Linear Data Structure	53
6.1	Array	53
6.1.1	Python Built-in Sequence: List, Tuple, String, and Range	55
6.1.2	Bonus	62
6.1.3	LeetCode Problems	62
6.2	Linked List	64
6.2.1	Singly Linked List	64
6.2.2	Doubly Linked List	67
6.2.3	Bonus	68
6.2.4	LeetCode Problems	70
6.3	Stack and Queue	71
6.3.1	Basic Implementation	72
6.3.2	Deque: Double-Ended Queue	74
6.3.3	Python built-in Module: Queue	76
6.3.4	Monotone Stack	77
6.3.5	Bonus	82
6.3.6	LeetCode Problems	83
6.4	Hash Table	84
6.4.1	Hash Function Design	86
6.4.2	Collision Resolution	87
6.4.3	Implementation	89
6.4.4	Python Built-in Data Structures	92

6.4.5	LeetCode Problems	94
7	Graphs and Trees	97
7.1	Graphs	97
7.1.1	Graph Representation	98
7.1.2	Types of Graph	100
7.2	Tree	100
7.2.1	Definition of Tree	101
7.2.2	Properties	101
7.3	Binary Tree	102
7.4	Binary Search Tree	105
7.4.1	Binary Searching Tree	106
7.5	Segment Tree	114
7.6	Trie for String	118
8	Heap and Priority Queue	127
8.1	Heap	127
8.1.1	Basic Implementation	128
8.1.2	Python Built-in Library: heapq	131
8.2	Priority Queue	132
8.3	Bonus	135
8.4	LeetCode Problems	136
IV	Complete Search	137
9	Linear Data Structures-based Search	141
9.1	Linear Search	141
9.2	Binary Search	142
9.2.1	Standard Binary Search and Python Module bisect .	142
9.2.2	Binary Search in Rotated Sorted Array	145
9.2.3	Binary Search on Result Space	147
9.2.4	LeetCode Problems	149
9.3	Two-pointer Search	152
9.3.1	Slow-fast Pointer	153
9.3.2	Opposite-directional Two pointer	156
9.3.3	Sliding Window Algorithm	157
9.3.4	LeetCode Problems	164
10	Elementary Graph-based Search	165
10.1	Graph Traversal	165
10.1.1	Breadth-First-Search (BFS)	166
10.1.2	Bidirectional Search: Two-end BFS	168
10.1.3	Depth-First-Search (DFS)	169

10.1.4 Backtracking	171
10.1.5 Summary	178
10.2 Tree Traversal	178
10.2.1 Depth-First-Search Tree Traversal Implementation . .	179
10.2.2 Level Order Tree Traversal Implementation	182
10.2.3 Time complexity of Binary Tree	182
10.2.4 Exercise	182
10.3 Exercise	183
10.3.1 Backtracking	183
11 Advanced Graph-based Search	185
11.1 Connected Components in Graph	185
11.1.1 In Undirected Graph	185
11.1.2 Directed Graph	190
11.2 Cycle Detection	190
11.3 Topological Sort	193
11.4 Minimum Spanning Trees	194
11.4.1 Kruskal and Prim Algorithm	194
11.5 Single-Source Shortest Paths	194
11.5.1 The Bellman-Ford Algorithm	194
11.5.2 Dijkstra's Algorithm	194
11.6 All-Pairs Shortest Paths	194
11.6.1 The Floyd-Warshall Algorithm	194
V Advanced Algorithm Design	195
12 Dynamic Programming	199
12.1 From Divide-Conquer to Dynamic Programming	201
12.1.1 Fibonacci Sequence	202
12.1.2 Longest Increasing Subsequence	203
12.2 Dynamic Programming Knowledge Base	207
12.2.1 Two properties	207
12.2.2 Four Elements	208
12.2.3 Dos and Do nots	208
12.2.4 Generalization: Steps to Solve Dynamic Programming	209
12.3 Problems Can be Optimized using DP	210
12.3.1 Example of optimizing $O(2^n)$ problem	210
12.3.2 Example of optimizing $O(n^3)$ problem	213
12.3.3 Single-Choice and Multiple-Choice State	218
12.4 Time Complexity Analysis	220
12.5 Exercises	222
12.6 Summary	232

13 Greedy Algorithms	235
13.1 From Dynamic Programming to Greedy Algorithm	235
13.2 Hacking Greedy Algorithm	236
13.2.1 Greedy-choice Property	236
VI Math and Bit Manipulation	239
14 Sorting and Selection Algorithms	241
14.1 $O(n^2)$ Sorting	243
14.1.1 Insertion Sort	243
14.1.2 Bubble Sort and Selection Sort	245
14.2 $O(n \log n)$ Sorting	248
14.2.1 Merge Sort	248
14.2.2 HeapSort	250
14.2.3 Quick Sort and Quick Select	250
14.3 $O(n + k)$ Counting Sort	254
14.4 $O(n)$ Sorting	256
14.4.1 Bucket Sort	256
14.4.2 Radix Sort	256
14.5 Lexicographical Order	259
14.6 Python Built-in Sort	260
14.6.1 Basic and Comparison	260
14.6.2 Customize Comparison Through Key	261
14.7 Summary and Bonus	263
14.8 LeetCode Problems	264
15 Bit Manipulation	271
15.1 Python Bitwise Operators	271
15.2 Python Built-in Functions	273
15.3 Twos-complement Binary	274
15.4 Useful Combined Bit Operations	276
15.5 Applications	278
15.6 LeetCode Problems	283
16 Math and Probability Problems	285
16.1 Numbers	285
16.1.1 Prime Numbers	285
16.1.2 Ugly Numbers	287
16.1.3 Combinatorics	289
16.2 Intersection of Numbers	292
16.2.1 Greatest Common Divisor	292
16.2.2 Lowest Common Multiple	293
16.3 Arithmetic Operations	294

16.4	Probability Theory	295
16.5	Linear Algebra	296
16.6	Geometry	296
16.7	Miscellaneous Categories	298
16.7.1	Floyd's Cycle-Finding Algorithm	298
16.8	Exercise	299
16.8.1	Number	299
VII	Appendix	301
17	Python Knowledge Base	303
17.1	Python Overview	304
17.1.1	Understanding Object	304
17.1.2	Python Components	305
17.2	Data Types and Operators	308
17.2.1	Arithmetic Operators	308
17.2.2	Assignment Operators	308
17.2.3	Comparison Operators	309
17.2.4	Logical Operators	309
17.2.5	Special Operators	310
17.3	Function	310
17.3.1	Python Built-in Functions	310
17.3.2	Lambda Function	311
17.3.3	Map, Filter and Reduce	311
17.4	Class	313
17.4.1	Special Methods	313
17.4.2	Class Syntax	314
17.4.3	Inheritance	314
17.4.4	Nested Class	314
17.5	Shallow Copy and the deep copy	315
17.5.1	Shallow Copy using Slice Operator	315
17.5.2	Deep Copy using copy Module	316
17.6	Global Vs nonlocal	317
17.7	Loops	317
17.8	Special Skills	318
17.9	Supplemental Python Tools	319
17.9.1	Re	319
17.9.2	Bitsect	319
17.9.3	collections	320

List of Figures

1	The connection between algorithms	4
1.1	Use category tag to "focus"	14
1.2	Use Test Case to debug	14
1.3	Use Test Case to debug	15
2.1	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.)	19
2.2	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.	20
2.3	Graphical examples for asymptotic notations.	22
2.4	Complexity Chart	23
2.5	Complexity of Common Data structures	25
3.1	Iteration vs recursion: in recursion, the line denotes the top-down process and the dashed line is the bottom-up process.	31
3.2	Call stack of recursion function	32
4.1	Divide and Conquer Diagram	35
4.2	Two Types of Recurrence Functions	37
5.1	The process to construct a recursive tree for $T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$	45

5.2 The cheat sheet for time and space complexity with recurrence function. If $T(n) = T(n-1)+T(n-2)+\dots+T(1)+O(n-1) = 3^n$	46
6.1 Array Representation	54
6.2 Linked List Structure	64
6.3 Doubly Linked List	67
6.4 Stack VS Queue	71
6.5 The process of decreasing monotone stack	78
6.6 Example of Hashing Table	85
6.7 The Mapping Relation of Hash Function	86
6.8 Hashtable chaining to resolve the collision	88
7.1 Four ways of graph representation	99
7.2 Example of a Tree with height and depth denoted	102
7.3 Example of Binary search tree of depth 3 and 8 nodes.	106
7.4 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.	108
7.5 Illustration of Segment Tree.	116
7.6 Trie VS Compact Trie	119
7.7 Trie Structure	120
8.1 Max-heap be visualized with binary tree structure on the left, and implemnted with Array on the right.	128
9.1 Example of Rotated Sorted Array	146
9.2 Two pointer Example	153
9.3 Slow-fast pointer to find middle	153
9.4 Floyd's Cycle finding Algorithm	155
9.5 One example to remove cycle	155
9.6 Sliding Window Algorithm	157
9.7 The array and the prefix sum	159
10.1 BFS VS DFS	165
10.2 Example Graph	165
10.3 Tree of possibilities for a typical backtracking algorithm . .	172
10.4 The state transfer graph of Combination and Permutation .	172
10.5 Example sudoko puzzle and its solution	176
10.6 Binary Tree	179
11.1 The connected components in undirected graph	186
11.2 Topological sort	193

12.1	Dynamic Programming Chapter Recap	199
12.2	Fibonacci number's Recursion Tree	200
12.3	Subproblem Graph for Fibonacci Sequence till n=4.	200
12.4	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.	204
12.5	The solution to LIS.	206
12.6	DP Decision	209
12.7	State Transfer for the panlindrom splitting	217
12.8	Summary of different type of dynamic programming problems	233
13.1	Screenshot of Greedy Catalog, showing the frequency and dif- ficulty of this type in the real coding interview	237
14.1	The whole process for insertion sort	244
14.2	One pass for bubble sort	245
14.3	The whole process for Selection sort	247
14.4	Merge Sort: divide process	249
14.5	Merge Sort: merge process	249
14.6	Hoarse Partition	252
14.7	Lomuto's Partition	269
14.8	The time complexity for common sorting algorithms	270
15.1	Two's Complement Binary for Eight-bit Signed Integers.	274
16.1	Example of floyd's cycle finding	298
17.1	Copy process	315
17.2	Caption	316
17.3	Caption	317
17.4	Caption	317

List of Tables

1.1	10 Main Categories of Problems on LeetCode, total 877 . . .	11
1.2	Problems categorized by data structure on LeetCode, total 877	13
1.3	10 Main Categories of Problems on LeetCode, total 877 . . .	13
2.1	Explanation of Common Growth Rate	24
6.1	Common Methods for Sequence Data Type in Python	55
6.2	Common Methods for Sequence Data Type in Python	55
6.3	Common Methods of List	57
6.4	Common Methods of String	59
6.5	Common Boolean Methods of String	59
6.6	Methods of Tuple	61
6.7	Common Methods of Deque	75
6.8	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.	76
6.9	Methods for Queue's three classes, here we focus on single-thread background.	76
7.1	Time complexity of operations for BST in big O notation . .	114
8.1	Methods of heapq	131
9.1	Methods of bisect	144
17.1	Arithmetic operators in Python	308

17.2 Comparison operators in Python	309
17.3 Logical operators in Python	310
17.4 Identity operators in Python	310
17.5 Membership operators in Python	310
17.6 Special Methods for Object Creation, Destruction, and Rep-	
resentation	313
17.7 Special Methods for Object Creation, Destruction, and Rep-	
resentation	314
17.8 Container Data types in collections module.	320

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 interview", 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 which is written by a person who has gone through the whole suffering stage of interview preparation, can ease your interview and provide one-stop information and preparation for your "coding interview". We integrate seamlessly of the online resource about interview process, the algorithm concepts from *Introduction to Algorithms*, the real interviewing problems from 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 (Section 4).

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.

Therefore, the purpose of this book is two-fold: to answer your questions about interview process and prepare you fully for the "coding interview" part.

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

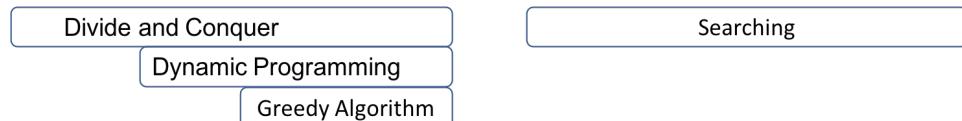


Figure 1: The connection between algorithms

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/>

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

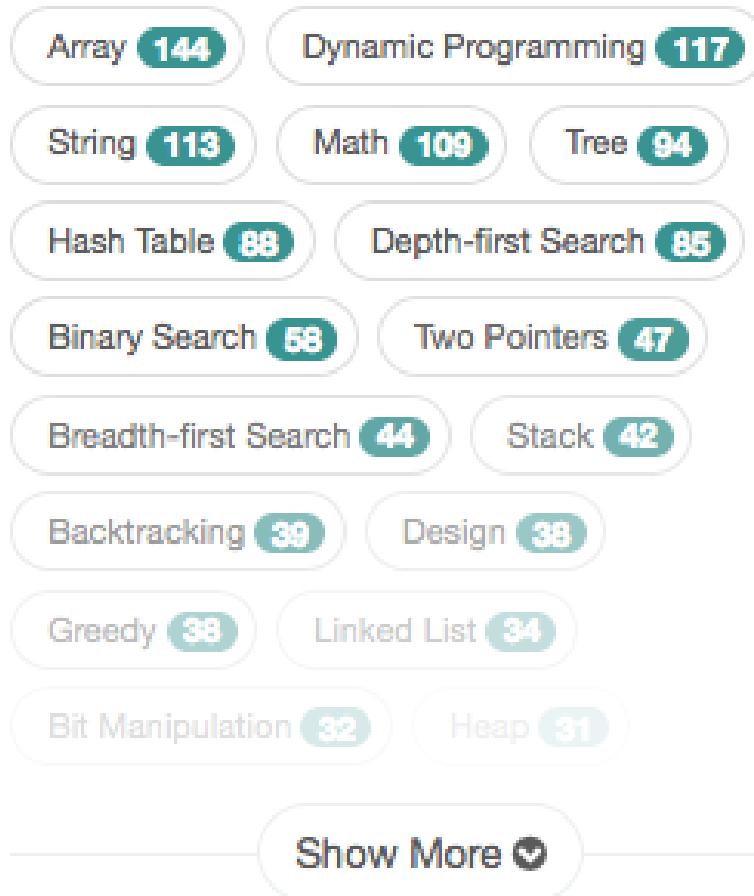


Figure 1.1: Use category tag to "focus"



Figure 1.2: Use Test Case to debug

The screenshot shows the discussion page for LeetCode problem 630. Course Schedule III. The page has a header with tabs: Description, Hints, Submissions, Discuss (which is selected), and Solution. Below the tabs is a search bar and a sort button labeled "New". There are four posts listed:

- A simple C solution 44ms**
Created at: May 11, 2018 1:14 AM | No replies yet.
VOTES 0 | VIEWS 62
- Sort in nlg n, DP using same method as Find Longest Increasing Subsequence**
Created at: April 5, 2018 5:15 AM | No replies yet.
VOTES 0 | VIEWS 206
- Simple Python code using priority queue with explanation**
Created at: November 5, 2017 7:36 AM | No replies yet.
VOTES 0 | VIEWS 205
- C++ priority queue solution**
Created at: October 8, 2017 11:42 PM | No replies yet.
VOTES 0 | VIEWS 152

Figure 1.3: Use Test Case to debug

2

Data Structures and Algorithms

“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

Data Structure is a way of collecting and organising 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 are normally written with programming language and are used to seek answers for real-world problems. 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, namely Complete Search in Part IV, Divide and Conquer (Part II), Dynamic Programming and

Greedy (Part V). In Section 2.1, 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 2.3.4.

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 2.3.

2.1 Four Problem Solving 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. Complete Search
2. Divide and Conquer
3. Dynamic Programming
4. Greedy Algorithm

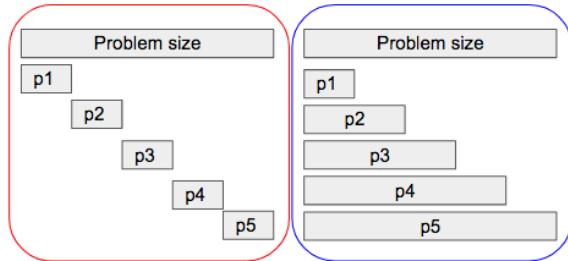


Figure 2.1: The dividing of problems of Divide and Conquer VS Dynamic programming. (Note: the left side in the red box is the Divide and Conquer, and the blue box is the dynamic programming.)

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 2.1, 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 2.1, 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 10.1.4, 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.

2.2 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

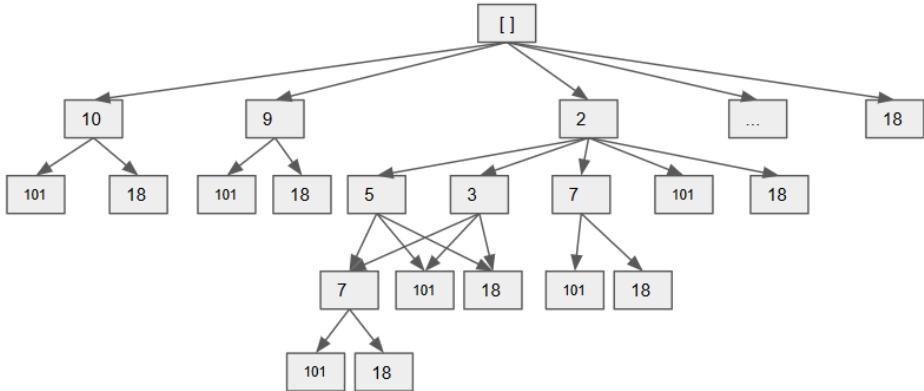


Figure 2.2: State Transfer Tree Structure for LIS, each path represents a possible solution. Each arrow represents a move: find an element in the following elements that's larger than the current node.

model this LIS problem, including:

1. Model the problem as a directed graph, where each node is the elements of the array, and an edge μ to v means node $v > \mu$. The problem now becomes finding the longest path from any node to any node in this directed graph.
2. Model the problem as a tree. The tree starts from empty root node, at each level i , the tree has $n-i$ possible children: $\text{nums}[i+1], \text{nums}[i+2], \dots, \text{nums}[n-1]$. There will only be an edge if the child's value is larger than its parent. Or we can model the tree as a multi-choice tree: for

combination problem, each element can either be chosen or not chosen. We would end up with two branch, and the nodes would become a path of the LIS, therefore, the longest LIS exist at the leaf nodes which has the longest length.

3. Model it with divide and conquer and optimal substructure.

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.

2.3 Complexity Analysis

The analysis of algorithms is used to predict the resources that the algorithm requires. 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. By analyzing several candidate algorithms for a problem, we can identify a most efficient one. Most books about algorithms, we assume a generic one-processor, with random-access machine (RAM) model of computation as our implementation technology which means the instructions are executed one after another, with no concurrent operations.

Usually we do not need to get the exact running time, because the extra precision is not usually worth of the effort of computing it. We only look at cases when the input sizes large enough to make only the order of growth of the running time relevant, so that we only need to study the asymptotic efficiency of algorithms. Thus, usually an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs. While, this book is more about practical usage of algorithms, which should be very different from the book introduction to algorithms. So we only introduce the basic concepts.

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

2.3.1 Asymptotic Notations

To analyze the complexity of an algorithm in each space or time, we have three ways: to give the upper bound, noted as O (pronounced as “big oh”), the lower bound, noted as Ω (pronounced as “big omega”), and a tight bound, noted as Θ (pronounced as “big theta”). It is easier to visualize it. All of these notations are applied to functions. For example, if the input size

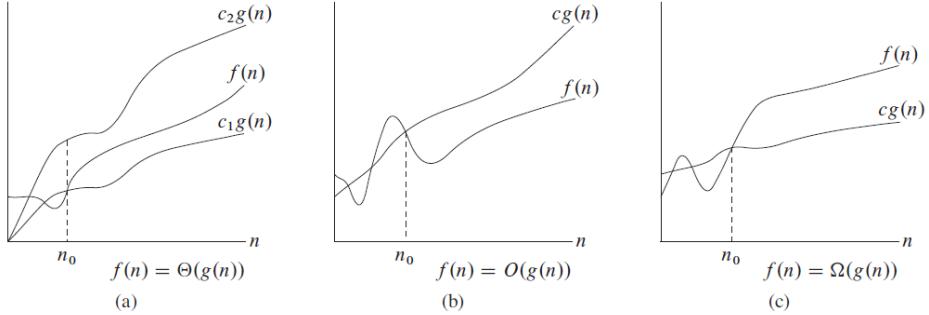


Figure 2.3: Graphical examples for asymptotic notations.

is n , then we can have functions like $an + b$, or $an^2 + bn + c$. But normally, we would only get the highest order of function, and simplified them to n and n^2 , with different notation. However, in the practical interviews, when the interviewer asks you to give the time and space complexity, you do not necessarily to give them the answer for each notation, you can just use O to denote, with regarding to different cases introduced in the next section.

2.3.2 Three Cases

We can convert these notations to more specific ways, two extreme ways: extremely bad or extremely good, and expected case.

1. Expected Case: which is the case that the data is not distributed in an extreme way as before, then this is used to denote the complexity for a normal cases. The expected case the time complexity we get can be represented as Θ .
2. Best Case: in the best scenario, the data is arranged in a way, that your algorithms run least amount of time. When we are analyzing the complexity of algorithms, to think of the best case can help us come up with a very reasonable lower bound, Ω .
3. Worst Case: in the worst scenario, its the opposite. This gave us a way to measure the upper bound of the complexity, which is denoted as O

2.3.3 Time and 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 if especially useful here. Normally we want to decrease the time complexity by sacrificing more space if the extra space is not a problem for the physical machine. But in some cases, decrease the time complexity is more important and needed, thus we need might go for alternative algorithms that uses less space but might with more time complexity.

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.

2.3.4 Big-O Cheat Sheet

In this section, we provide the plotting of common seen time complexity functions (shown in Fig 2.4): 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/>.

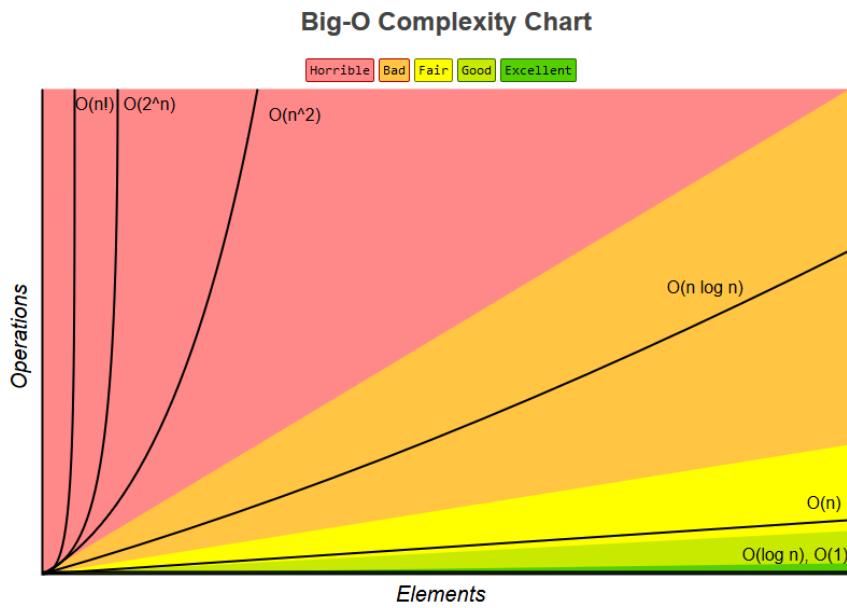


Figure 2.4: Complexity Chart

Also, we provide the average and worst time and space complexity for the some classical data structure's operations (shown in Fig. 2.5) and of

Table 2.1: 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

algorithms (shown in Fig. 2.5).

Common Data Structure Operations

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

Array Sorting Algorithms

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

Figure 2.5: Complexity of Common Data structures

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

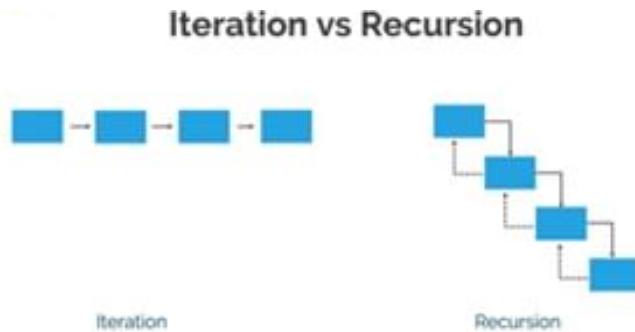


Figure 3.1: Iteration vs recursion: in recursion, the line denotes the top-down process and the dashed line is the bottom-up process.

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

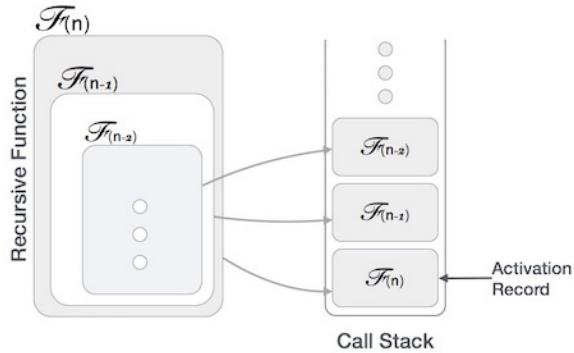


Figure 3.2: Call stack of recursion function

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

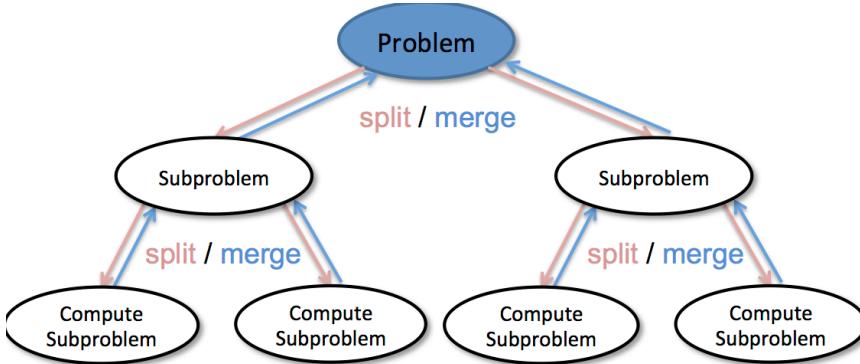


Figure 4.1: Divide and Conquer Diagram

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 have two different ways to divide a problem into subproblems, and we use **recurrence function** and **recursion tree** for distinguishment. $f(n)$ is a function which denotes the time cost of dividing and combining step together of the problem with size n :

1. **Non-overlapping subproblems** where our subproblems are disjoint with each other

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

Or in Recursion calls, where $F(n)$ is the solution to problem of size n , and *Comb* refers to function or programming operations to combine the solutions to subproblems in order to obtain the solution to current problem.

$$F(n) = \text{Comb}(F(0, n/b), F(n/b + 1, 2n/b), \dots, F(n - n/b, n)) \quad (4.2)$$

where $a \leq 1, b > 1$, where the divide-and-conquer algorithm creates a non-overlapping subproblem each with size n/b .

2. **Overlapping subproblems** 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)

```

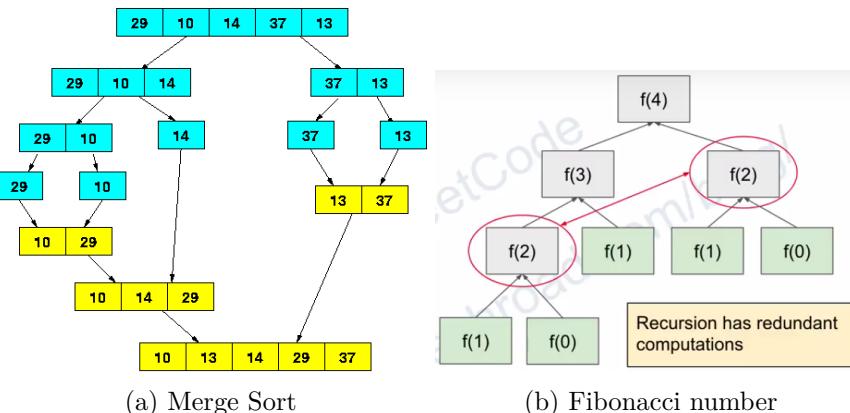


Figure 4.2: Two Types of Recurrence Functions

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 12).

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 14);
- Binary Search (Section 9.2);
- Heap(Section 8.1);
- Binary Search Tree (Section 7.4);
- Segment Tree(Section 7.5).

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. $[0, -2, -1, -4, 0, -1, 1, 2, -3, 1]$, $\Rightarrow O(n)$, then we use prefix_sum, the difference is we set prefix_sum to 0 when it is smaller than 0, $O(n)$

```
1 from    sys import maxint
2 class Solution( object ):
3     def maxSubArray( self ,  nums):
4         """
5             :type nums: List[ int ]
6             :rtype: int
7         """
8         max_so_far = -maxint - 1
9         prefix_sum= 0
10        for i in range(0, len( nums )):
11            prefix_sum+= nums[ i ]
12            if (max_so_far < prefix_sum):
13                max_so_far = prefix_sum
14
15            if prefix_sum< 0:
16                prefix_sum= 0
17        return max_so_far
```


5

Algorithm Analysis

In this chapter, we centralize on the techniques to analyze the complexity—mainly average and/or worst time and space – of an algorithm we designed. We consider that the rules and methods are quite universal. We includes three parts:

1. Time Complexity (Sec. ??): first for simple iteration programming, then to analyze the recurrent function of: divide-conquer and dynamic programming.
2. Space Complexity (Sec. ??): space complexity is usually way more straightforward compared with time complexity.
3. Amortized Analysis (Sec. ??).

5.1 Time Complexity

Simple example These are just straightforward for us to analyze the running time. Sometimes, things become more obscure. Then, we need more advanced techniques to help us handle. For example, we use recurrence function to represent the the time we need when the problem decrease the size. Such that for one for loop, we can use $T(n) = T(n - 1) + O(1)$, and for two nested for loops, normally $T(n) = T(n - 1) + O(n)$ is enough to represent this situation. For a divide and conquer problem, we might get a recurrence function as $T(n) = T(n/2) + O(1)$. With recurrence function, the time complexity analysis is conveniently converted to a math problem and things get to be more interesting. We can divide the recurrence function into two types: non-overlapping as $T(n) = T(a * n/b) + f(n)$, and with over-lapping as $T(n) = T(a * n/b) + f(n)$.

5.1.1 Solve Recurrence Function

The complexity analysis includes the time and space complexity. While for the divide and conquer methods, because of the usage of the recursion function, it could be slightly tricky and we need to learn how to attack this complexity. Also it is usually required and asked by interviewers in real interview.

5.1.2 Time Complexity

The core to analyze the time complexity of the divide and conquer methodology is by characterizing the recurrence relation shown in Eq. 14.2. In general we have three ways to do this, 1) substitution method; 2) recursion-tree method; 3) master method. However, in real situation, it depends on XX to choose which one to use. Each one has its own limitation. In this book, we prove enough theory about how to compute the time complexity for the recurrence equation for practical coding or interview situation. If you want to learn more, it is a good choice to refer the book (Introduction to Algorithms). Here, we wont detail on the substitution method, because in real interviews, we need some quick and more straightforward ways to get the computational cost. And the substitution method is more used to prove the cost in a very rigorous way. Recursion tree and the master therorems are the main ways we rely on to answer the time complexity for a divide and conquer method shown in Eq. 14.2.

Recursion Tree Method

Drawing out a recursion tree serves as a straightforward way to come up with a good guess. Normally we can tolerate a small amount of "sloppiness", because later on, we can prove the complexity with substitution method discussed in the last section. However, when we are drawing the recursion tree, if we are careful enough and summing up the costs from each level and each node, we can use is as a direct proof of the solution to the recurrence.

In the corresponding recursion tree for recurrence equation in divide and conquer, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion. Let's look at one example for given recursion $T(n) = 3T([n/4]) + \Theta(n^2)$. We replace $\Theta(n^2) = cn^2$, where $c > 0$. cn^2 is the cost we pay to divide a problem with n input size to three problems each with $n/4$ input size and combine the solution of the subproblems to solve the current problem. We first expand $T(n)$, and put the cost cn^2 at the root, and with three children each noted with $T(n/4)$. Then we recursively replace $T(n/4)$ with the cost and its subproblem till the size of each subproblem to be 1, which means we

get to the leaves. The computational complexity for this recursion would be the sum of all layers's costs. And we assume $T(1) = 1$.

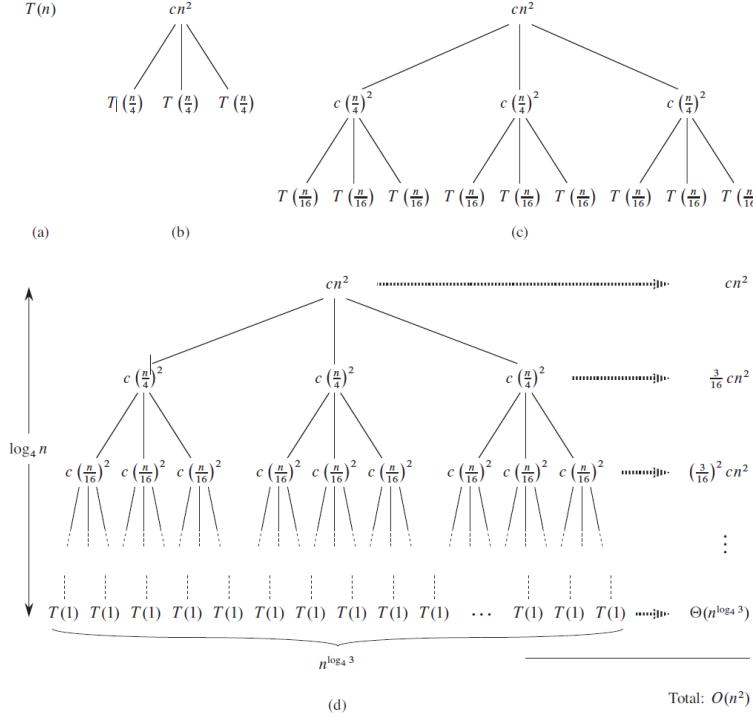


Figure 5.1: The process to construct a recursive tree for $T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$

$$\begin{aligned}
 T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2cn^2 + \dots + (\frac{3}{16})^{\log_4 n - 1}cn^2 + \Theta(n^{\log_4 3}) \\
 &= \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \\
 &< \sum_{i=0}^{\infty} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
 &= O(n^2).
 \end{aligned} \tag{5.1}$$

Master Method

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:

For Eq. 14.2, let $a \geq 1, b > 1$, we first compute $n^{\log_b a}$,

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 $T(n) = \Theta(f(n))$.

5.2 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.

5.2.1 Summary

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

Equation	Time	Space	Examples
$T(n) = 2*T(n/2) + O(n)$	$O(n\log n)$	$O(\log n)$	quick_sort
$T(n) = 2*T(n/2) + O(n)$	$O(n\log n)$	$O(n + \log n)$	merge_sort
$T(n) = T(n/2) + O(1)$	$O(\log n)$	$O(\log n)$	Binary search
$T(n) = 2*T(n/2) + O(1)$	$O(n)$	$O(\log n)$	Binary tree traversal
$T(n) = T(n-1) + O(1)$	$O(n)$	$O(n)$	Binary tree traversal
$T(n) = T(n-1) + O(n)$	$O(n^2)$	$O(n)$	quick_sort(worst case)
$T(n) = n * T(n-1)$	$O(n!)$	$O(n)$	permutation
$T(n) = T(n-1)+T(n-2)+...+T(1)$	$O(2^n)$	$O(n)$	combination

Figure 5.2: The cheat sheet for time and space complexity with recurrence function. If $T(n) = T(n-1)+T(n-2)+...+T(1)+O(n-1) = 3^n$

5.2.2 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, $O(\log n)$.

```

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)

```

5.3 Amortized Analysis

Part III

Footstone: Data Structures

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 varieties 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 go 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 categorized as two main types: *Liner* (Chapter 6) and *Non-liner* (include: Heap and Graph in Chapter 7 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

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.

6.1 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. 6.1. 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.*

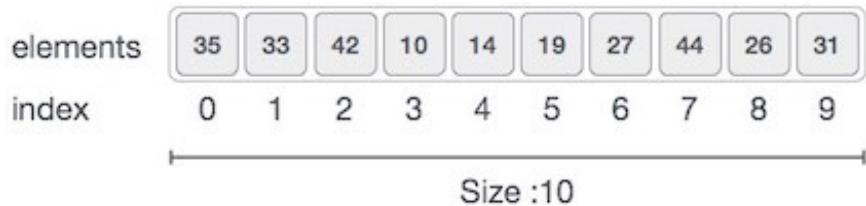


Figure 6.1: Array Representation

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.

6.1.1 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 6.1 and 6.2. To note that in Python, the indexing starts from 0.

Table 6.1: 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 6.2: 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 6.3 shows us the common List Methods, and they will be used as list.methodName().

Table 6.3: 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 s
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 6.4 and 6.5 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.

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:

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 6.4: 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 6.5: 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

```

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 6.6.

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:

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.

Table 6.6: 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

- 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

```

6.1.2 Bonus

Circular Array The corresponding problems include:

1. 503. Next Greater Element II

6.1.3 LeetCode Problems

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
4      dog","a8 act zoo"]
5 Output: ["g1 act car","a8 act zoo","ab1 off key dog","a1 9
6      2 3 1","zo4 4 7"]
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(( ' '.join(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

```

6.2 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.

6.2.1 Singly Linked List

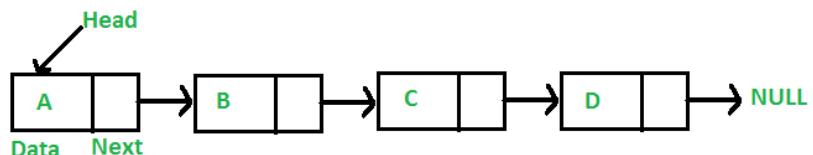


Figure 6.2: Linked List Structure

Fig. 6.2 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 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 to set the head to None

```

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

```

6.2.2 Doubly Linked List

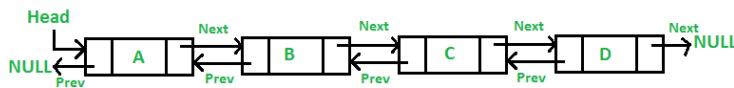


Figure 6.3: Doubly Linked List

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.

6.2.3 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.

6.1 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.

```

# use of dummy node
def iterative(head):
    dummy = ListNode(None)
    dummy.next = head
    current = dummy
    while current.next:
        # current pointer wont move unless the next has
        # different value
        if current.val == current.next.val:
            # delete next
            current.next = current.next.next
        else:
            current = current.next
    return head

```

Solution 2: Recursion.

```

def recursive(node):
    if node.next is None:
        return node
    next = recursive(node.next)
    if next.val == node.val:
        # delete next
        node.next = node.next.next
    return node

```

6.2.4 LeetCode Problems

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)

6.3 Stack and Queue

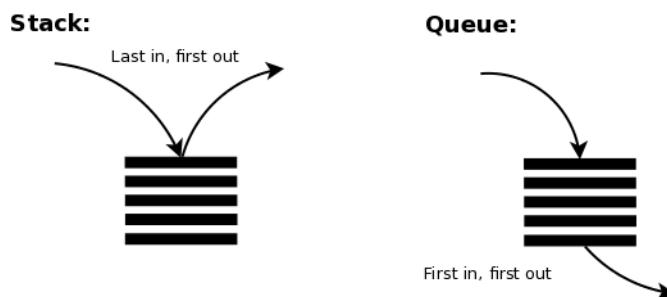


Figure 6.4: Stack VS Queue

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 officialized 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 10, (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 10, (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.

6.3.1 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.

6.3.2 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 6.7.

Table 6.7: 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.

6.3.3 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 6.8. *In python 3, we use lower case queue, but in Python 2.x it uses Queue, in our book, we learn Python 3.*

Table 6.8: 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 6.9.

Table 6.9: 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
```

6.3.4 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. 6.5. *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:

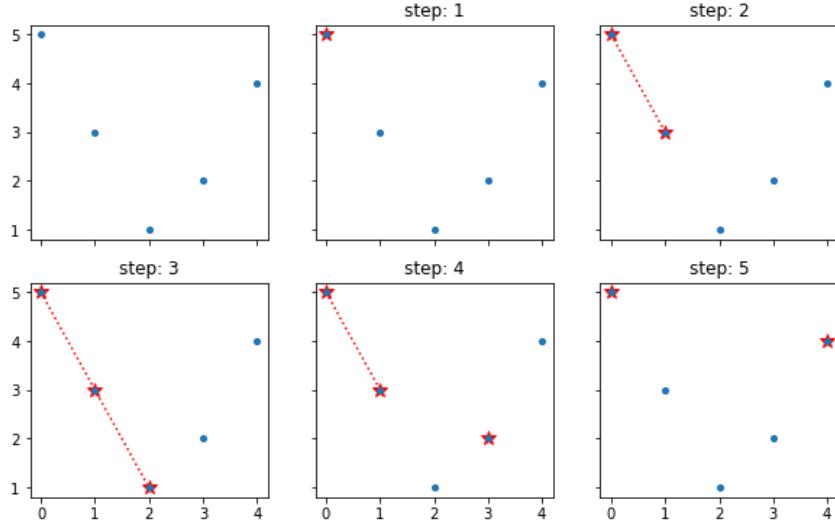


Figure 6.5: The process of decreasing monotone stack

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

```

6.2 907. 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)

```

6.3.5 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.

6.3.6 LeetCode Problems

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"],
16       inputs = [[], [1], [100], [3001], [3002]]
17 Output: [null,1,2,3,3]
```

Analysis: This is a typical buffer problem. If the size is larger than the buffer, then we squeeze out the easiest data. Thus, a queue can be used to save the t and each time, squeeze any time not in the range of [t-3000, t]:

```

1 class RecentCounter:
2
3     def __init__(self):
4         self.ans = collections.deque()
```

```

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

6.4 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

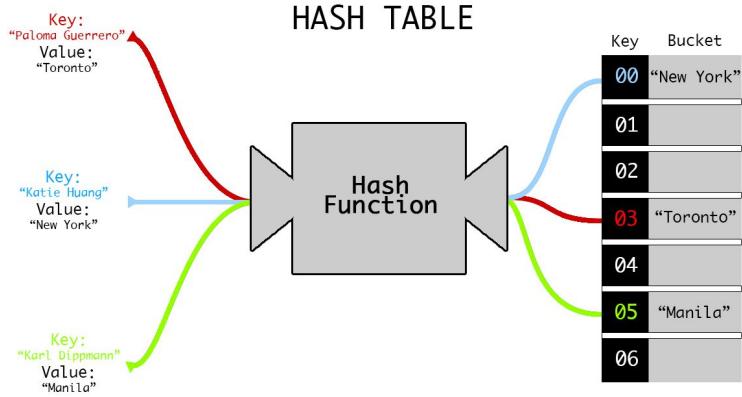


Figure 6.6: Example of Hashing Table

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. 6.6 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.

6.4.1 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.

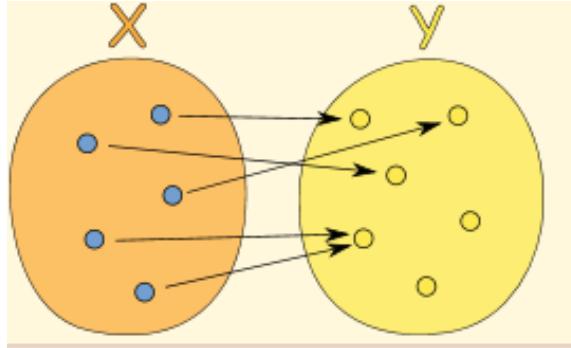


Figure 6.7: The Mapping Relation of Hash Function

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. 6.7. 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

6.4.2 Collision Resolution

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

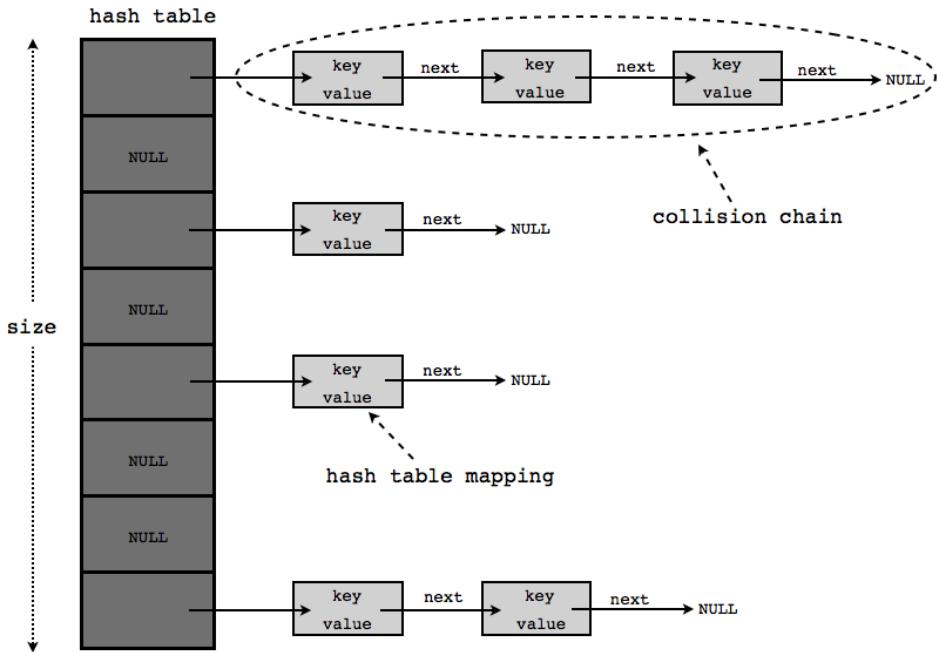


Figure 6.8: Hashtable chaining to resolve the collision

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. 6.8.

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

6.4.3 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

```

6.4.4 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

dict Data Type 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 `set` and `set()` with a single word argument.

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 orderedDict 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

6.4.5 LeetCode Problems

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,
   separated by the @ sign.
2
3 For example, in alice@leetcode.com, alice is the local name, and
   leetcode.com is the domain name.
4
5 Besides lowercase letters, these emails may contain '.'s or '+'s
   .
6
7 If you add periods ('.') between some characters in the local
   name part of an email address, mail sent there will be
   forwarded to the same address without dots in the local name.
   For example, "alice.z@leetcode.com" and "alicez@leetcode.
   com" forward to the same email address. (Note that this rule
   does not apply for domain names.)
8
9 If you add a plus ('+') in the local name, everything after the
   first plus sign will be ignored. This allows certain emails
   to be filtered, for example m.y+name@email.com will be
   forwarded to my@email.com. (Again, this rule does not apply
   for domain names.)

```

```

10 It is possible to use both of these rules at the same time.
11
12 Given a list of emails, we send one email to each address in the
13 list. How many different addresses actually receive mails?
14
15 Example 1:
16
17 Input: ["test.email+alex@leetcode.com", "test.e.mail+bob.
18           cathy@leetcode.com", "testemail+david@lee.tcode.com"]
18 Output: 2
19 Explanation: "testemail@leetcode.com" and "testemail@lee.tcode.
20           com" actually receive mails
21 Note:
22     1 <= emails[i].length <= 100
23     1 <= emails.length <= 100
24     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)

```

Graphs and Trees

7.1 Graphs

Graph is a widely used data structure to model real-world problems. Graph is a collection of *vertices* and *edges* (which connects two vertices). The *weights of edges* refers to the information with edges. In this section, we only introduce **graph representations** in Python and **graph types** in the following two subsections. We explore the important graph complete search methodologies: Searching Graphs in Chapter 10 and Shortest Path algorithms in Chapter ??.

We use G to denote the graph, V and E to refer its collections of vertices and edges, respectively. Before move to the main contents, there are some Python matrices skills we will cover here.

Python Matrices In order to support the Graph Representation section, we need to learn the following matrices operations:

New a 2-d Matrix and Modify Element:

```

1 rows, cols = 3, 3
2 nodes = 3
3 matrix1 = [[0 for _ in range(cols)] for _ in range(rows)] # rows
   * cols
4 matrix2 = [[] for _ in range(nodes)] # rows with empty list as
   elements
5 matrix3 = [[0]*cols for _ in range(rows)] # rows * cols
6 # assign elements
7 matrix1[1][2] = 2
8 matrix3[1][2] = 2
9 # output by printing
10 # [[0, 0, 0], [0, 0, 2], [0, 0, 0]]
```

```
11 # [[0, 0, 0], [0, 0, 2], [0, 0, 0]]
```

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

```
1 # wrong declaration
2 matrix4 = [[0]*cols]*rows
3 matrix4[1][2] = 2
4 # output by printing
5 # [[0, 0, 2], [0, 0, 2], [0, 0, 2]]
```

Accessing Rows:

```
1 # accessing row
2 for row in matrix1:
3     print(row)
4 # output
5 # [0, 0, 0]
6 # [0, 0, 2]
7 # [0, 0, 0]
```

Accessing Cols: 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. The speed is cols times slower than accessing rows.

```
1 # accessing col
2 for i in range(cols):
3     col = [row[i] for row in matrix1]
4     print(col)
5 # output
6 # [0, 0, 0]
7 # [0, 0, 0]
8 # [0, 2, 0]
```

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

```
1 transposedMatrix1 = list(zip(*matrix1))
2 print(transposedMatrix1)
3 # output
4 # [(0, 0, 0), (0, 0, 0), (0, 2, 0)]
```

7.1.1 Graph Representation

There are totally four ways to represent graph and store related edge information: 1) Adjacency Matrix, 2) Adjacency List, 3) Edge List, and 4) Tree Structure. An example is shown in Fig 7.1.

1. Adjacency Matrix Adjacency matrices are in the form of a 2-D matrix as we have shown at the beginning of this section. This is the simplest type of graph representation and for situation where we know total vertices V , the matrix is of size V^2 . For unweighted graph, we can that edge from

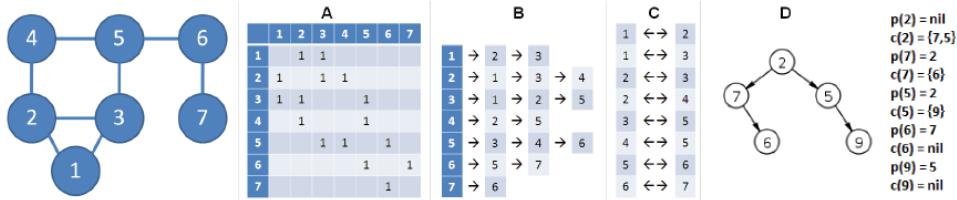


Figure 7.1: Four ways of graph representation

vertex i to j as $G[i][j] = 1$, and 0 otherwise. For a weighted graph it will be the weight(i, j) instead and 0 otherwise.

This type of representation usually fits to the dense graph where only small ratio of the matrix is blank. An adjacency matrix requires exactly $O(V^2)$ 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.

2. Adjacency List An Adjacency list is a more compact and efficient form of graph representation compared with 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, $[[1, 2, 3], [3, 1], [4, 6, 1]]$, node 0 connects to 1,2,3, node 1 connect to 3,1, node 2 connects to 4,6,1. We can declare the matrix as in Matrix2 shown in before.

Except for the basic list of list, if you need a “name” for each node, and if we have no knowledge of the number of nodes you might get, use Python dictionary instead. For example, we are given a list of edges $[["a", "c"], [b, c], [b, e], \dots]$. 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.

```

1 from collections import defaultdict
2     d = defaultdict(set) # structure initialization
3     # add an edge (remember to add to both vertices!)
4     for ver1, ver2 in edges:
5         d[ver1].add(ver2) #use add because it is set
6     #the following printed graph

```

And the printed graph is as follows:

```

graph = { "a" : [ "c" ] ,
          "b" : [ "c", "e" ] ,
          "c" : [ "a", "b", "d", "e" ] ,
          "d" : [ "c" ] ,
          "e" : [ "c", "b" ] ,
          "f" : []
}

```

If we need weights for each edge, use dictionary from the default dictionary to represent:

```

1 graph= collections.defaultdict(dict)
2 for (num, den), val in zip(equations, values):
3     graph[num][num] = graph[den][den] = 1.0
4     graph[num][den] = val
5     graph[den][num] = 1 / val
6 #defaultdict(<type 'dict'>, {u'a': {u'a': 1.0, u'b': 2.0}, u'c':
    {u'c': 1.0, u'b': 0.3333333333333333}, u'b': {u'a': 0.5, u'c':
        ': 3.0, u'b': 1.0}})
```

3. 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.

Function to generate the list of all edges from either Adjacency Matrix or Adjacency List is similar to the following code:

```

1 def generate_edges(graph):
2     edges = []
3     for node in graph:
4         for neighbour in graph[node]:
5             edges.append((node, neighbour))
6
7     return edges
8
9 print(generate_edges(graph))
```

4. 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. For each vertex, we only store two attributes: the parent and the list of children.

7.1.2 Types of Graph

7.2 Tree

Trees are well-known as a non-linear data structure. They organize data hierarchically other than in the linear way.

In the first section, we will introduce the definition and common properties of **tree** (Section ??). There are different types of trees devised for different purpose. 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. Therefore, in the second section, we talk about the **binary tree** (Section 7.3) which is the most common type of trees used to solve real-world problems. Then in the third section, we introduce **searching trees** (Section ??): we can see how such trees help us to store, and find the information we need. Mainly two types of searching trees are included: binary search tree which only has two branches and the trie which might have multiple and varied number of branches for each node. The last two section is the bonus section, with the more advanced tree structures *Trie* (Section 7.6) and *Segment Tree* (Section ??) for string pattern match and array queries. Compared with the previous sections, these two structures are not necessities but Trie does help improve the time complexity of string pattern matching.

Also, to be noticed, this chapter serves as an introduction, the summary and commonly seen problems related to tree will be detailed in Part ??.

7.2.1 Definition of Tree

A **tree** is a type of acyclic graph and it is defined as a collection of entities which are called **nodes**. Nodes are connected by **edges**. Each node contains a value or data, and it may or may not have a child node.

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. We can see the root, leave nodes in Fig 7.2.

A **path** is defined as a sequence of nodes and edges connecting a node with a descendant. We can classify them into three types:

1. Root->Leaf Path: the starting and ending node of the path is the root and leaf node respectively;
2. Root->Any Path: the starting and ending node of the path is the root and any node (Root, inner, leaf node) respectively;
3. Any->Any Path: the starting and ending node of the path is both any node (Root, inner, leaf node) respectively.

7.2.2 Properties

The property of a tree starts from the property of a *node* (shown in Fig 7.2).

1. The **depth** (or level) of a node is the number of edges from the node to the tree's root node. And it can be obtained from up-down level-by-level traversal.

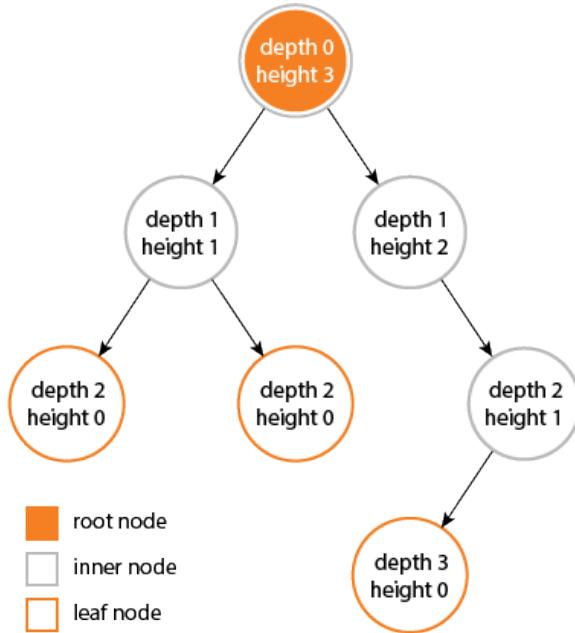


Figure 7.2: Example of a Tree with height and depth denoted

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

Properties of a *tree*:

1. The **height**(or **depth**) of a tree would be the height of its root node, or equivalently, the depth of its deepest node.
2. The **diameter** (or **width**) of a tree is the number of nodes (or edges) on the longest path between any two leaf nodes.

Forest is a set of $n \geq 0$ disjoint trees.

7.3 Binary Tree

A binary tree is made of nodes which has at most two branches—the “left child” and the “right child”—and a data element. The “root” node is the

topmost node in the tree. The left and right child recursively point to smaller "subtrees" on either side.

In Python, a tree node is implemented as:

```

1 class TreeNode:
2     def __init__(self, val):
3         self.val = val
4         self.left = None
5         self.right = None

```

A binary tree can be either implemented as *a tree or an array*.

- When using an array, The index relation between parent node and the child nodes are: idx for parent, the children will be $idx * 2 + 1$ and $idx * 2 + 2$ as demonstrated in Fig ??.
- To construct a tree from a list which use "None" to denote for empty node, which can be really useful to automatize the tree construction. We can use the following code:

```

1 def generateBinaryTree(nums, idx):
2     if idx >= len(nums):
3         return None
4     node = TreeNode(nums[idx])
5     node.left = generateBinaryTree(nums, idx*2+1)
6     node.right = generateBinaryTree(nums, idx*2+2)
7     return node
8
9 nums = [1, 2, 3, 4, 5, None, 6]
10 root = generateBinaryTree(nums, 0)

```

To show the nodes at each level, we use LevelOrder function to print out the tree:

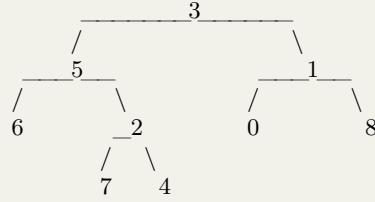
```

1 def LevelOrder(root):
2     q = [root]
3     while q:
4         new_q = []
5         for n in q:
6             if n is not None:
7                 print(n.val, end=', ')
8                 if n.left:
9                     new_q.append(n.left)
10                if n.right:
11                    new_q.append(n.right)
12        q = new_q
13        print('\n')
14 LevelOrder(root)
15 # output
16 # 1,
17
18 # 2,3,
19
20 # 4,5,None,6,

```

Lowest Common Ancestor. The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself). There will be two cases in LCA problem which will be demonstrated in the following example.

7.1 Lowest Common Ancestor of a Binary Tree (L236). Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree. Given the following binary tree: root = [3,5,1,6,2,0,8,null,null,7,4]



Example 1:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

Output: 5

Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

Solution: Divide and Conquer. There are two cases for LCA: 1) two nodes each found in different subtree, like example 1. 2) two nodes are in the same subtree like example 2. If we compare the current node with the p and q, if it equals to any of them, return current node in the tree traversal. Therefore in example 1, at node 3, the left return as node 5, and the right return as node 1, thus node 3 is the LCA. In example 2, at node 5, it returns 5, thus for node 3, the right tree would have None as return, thus it makes the only valid return as the final LCA. The time complexity is $O(n)$.

```

1 def lowestCommonAncestor(self, root, p, q):
2     """
3     :type root: TreeNode
4     :type p: TreeNode
5     :type q: TreeNode
6     :rtype: TreeNode
7     """
8     if not root:
9         return None
10    if root == p or root == q:
11        return root # found one valid node (case 1: stop at
12        # 5, case 2:stop at 5)
  
```

```

12     left = self.lowestCommonAncestor(root.left, p, q)
13     right = self.lowestCommonAncestor(root.right, p, q)
14     if left is not None and right is not None: # p, q in
15         the subtree
16         return root
17     if any([left, right]) is not None:
18         return left if left is not None else right
19     return None

```

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.

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: $L = Non-L + 1$.

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.

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.

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.

A degenerate (or pathological) tree A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

7.4 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, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSION, INSERT, and DELETE. Thus, a search tree can be both used as a dictionary and a priority queue.

In this section, we will introduce the most commonly used two types of searching trees: binary searching tree (BST) and Trie where the keys are usually numeric numbers and strings respectively.

7.4.1 Binary Searching Tree

A binary search tree (BST) is an organized searching tree structure in binary tree, as the name suggests. Binary search trees whose internal nodes each store a key (and optionally, an associated value), each node have two distinguished sub-trees (if only one sub-tree the other is None).

BST keep their keys in sorted order, so that lookup and other operations can use the *principle of binary search tree*:

Let x be a node in a binary search tree, if y is a node in the left subtree of x , them $y.key \leq x.key$. If y is a node in the right subtree of x , then $y.key \geq x.key$.

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 definiton. One example of BST without duplicates is shown in Fig 7.3.

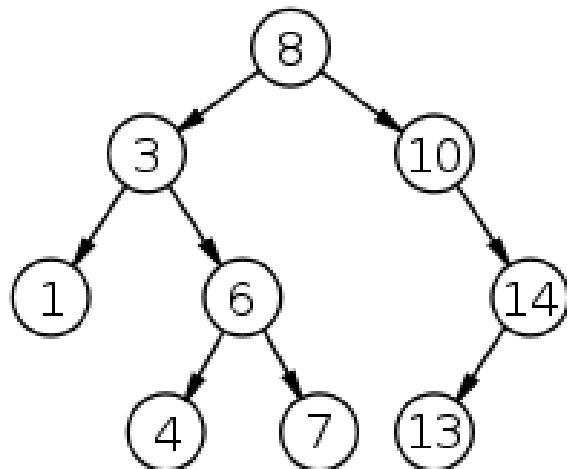
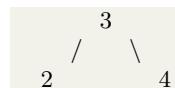


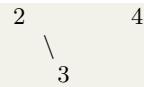
Figure 7.3: Example of Binary search tree of depth 3 and 8 nodes.

Solve Duplicate Problem 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:

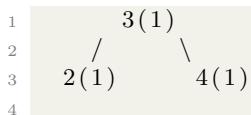




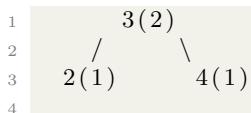
Note that the duplicates are not in contiguous levels.

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.

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.

SEARCH There are two different implementations for SEARCH: recursive and iterative.

```

1 # recursive searching
2 def search(root, key):
  
```

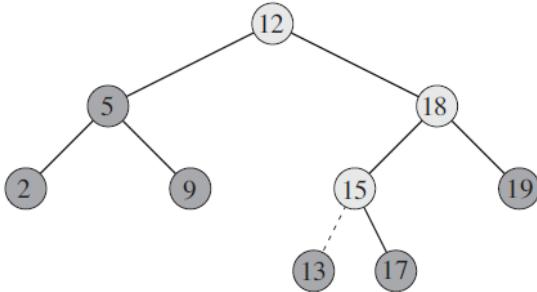


Figure 7.4: The lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

```

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 7.4. A new key is always inserted at leaf (there are other ways to insert but here we only discuss this one way). We start searching a key from root till we hit an empty node. Then we new a TreeNode and insert this new node either as the left or the child node according to the searching property. Here we still shows both the recursive and iterative solutions.

```

1 # Recursive insertion
2 def insertion(root, key):
3     if root is None:
4         root = TreeNode(key)
5         return root
6     if root.val < key:
  
```

```

7     root.right = insertion(root.right, key)
8 else:
9     root.left = insertion(root.left, key)
10    return root

```

The above code needs return value and reassign the value for the right and left every time, we can use the following code which might looks more complex with the if condition but works faster and only assign element at the end.

```

1 # recursive insertion
2 def insertion(root, val):
3     if root is None:
4         root = TreeNode(val)
5         return
6     if val > root.val:
7         if root.right is None:
8             root.right = TreeNode(val)
9         else:
10            insertion(root.right, val)
11    else:
12        if root.left is None:
13            root.left = TreeNode(val)
14        else:
15            insertion(root.left, val)

```

We can search the node iteratively and save the previous node. The while loop would stop when hit at an empty node. There will be three cases in the case of the previous node.

1. The previous node is None, which means the tree is empty, so we assign a root node with the value
2. The previous node has a value larger than the key, means we need to put key as left child.
3. The previous node has a value smaller than the key, means we need to put key as right child.

```

1 # iterative insertion
2 def iterativeInsertion(root, key):
3     pre_node = None
4     node = root
5     while node is not None:
6         pre_node = node
7         if key < node.val:
8             node = node.left
9         else:
10            node = node.right
11     # we reached to the leaf node which is pre_node
12     if pre_node is None:
13         root = TreeNode(key)
14     elif pre_node.val > key:
15         pre_node.left = TreeNode(key)
16     else:
17         pre_node.right = TreeNode(key)
18     return root

```

BST Generation First, let us declare a node as BST which is the root node. Given a list, we just need to call INSERT for each element. The time complexity can be $O(n \log n)$.

```

1 datas = [8, 3, 10, 1, 6, 14, 4, 7, 13]
2 BST = None
3 for key in datas:
4     BST = iterativeInsertion(BST, key)
5 print(LevelOrder(BST))
6 # output
7 # [8, 3, 10, 1, 6, 14, 4, 7, 13]
```

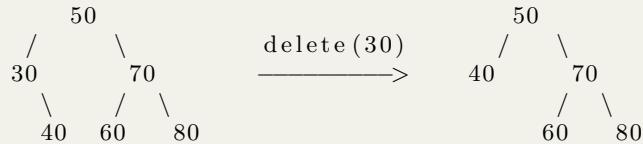
DELETE Before we start to check the implementation of DELETE, I would suggest the readers to read the next subsection—the Features of BST at first, and then come back here to finish this paragraph.

When we delete a node, three possibilities arise.

- 1) Node to be deleted is leaf: Simply remove from the tree.



- 2) Node to be deleted has only one child: Copy the child to the node and delete the child



- 3) Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

Features of BST

Minimum and Maximum The operation is similar to search, to find the minimum, we always traverse on the left subtree. For the maximum, we just need to replace the “left” with “right” in the key word. Here the time complexity is the same $O(lgn)$.

```

1 # recursive
2 def get_minimum(root):
3     if root is None:
4         return None
5     if root.left is None: # a leaf or node has no left subtree
6         return root
7     if root.left:
8         return get_minimum(root.left)
9
10 # iterative
11 def iterative_get_minimum(root):
12     while root.left is not None:
13         root = root.left
14     return root

```

Also, sometimes we need to search two additional items related to a given node: successor and predecessor. The structure of a binary search tree allows us to determine the successor or the predecessor of a tree without ever comparing keys.

Successor of a Node A successor of node x is the smallest item in the BST that is strictly greater than x . It is also called in-order successor, which is the next node in Inorder traversal of the Binary Tree. Inoreder Successor is None for the last node in inorder traversal. If our TreeNode data structure has a parent node.

Use parent node: the algorihtm has two cases on the basis of the right subtree of the input node.

For the right subtree of the node:

- 1) If it is not None, then the successor is the minimum node in the right subtree. e.g. for node 12, $\text{successor}(12) = 13 = \min(12.\text{right})$
- 2) If it is None, then the successor is one of its ancestors. We traverse up using the parent node until we find a node which is the left child of its parent. Then the parent node here is the successor. e.g. $\text{successor}(2)=5$

The Python code is provided:

```

1 def Successor(root, n):
2     # Step 1 of the above algorithm
3     if n.right is not None:
4         return get_minimum(n.right)
5     # Step 2 of the above algorithm
6     p = n.parent

```

```

7 while p is not None:
8     if n == p.left :# if current node is the left child node,
9         then we found the successor , p
10        return p
11    n = p
12    p = p.parent
13 return p

```

However, if it happens that your tree node has no parent defined, which means you can not traverse back its parents. We only have one option. Use the inorder tree traversal, and find the element right after the node.

For the right subtree of the node:

- 1) If it is not None, then the successor is the minimum node in the right subtree. e.g. for node 12, 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 node, 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 $O(n)$.

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]
2
3
4
5
6
7
8
9
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

```

```

graph TD
    6 --- 2
    6 --- 8
    2 --- 0
    2 --- 4
    4 --- 3
    4 --- 5
    8 --- 7
    8 --- 9

```

Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself 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

Now we put a table here to summarize the space and time complexity for each operation.

Table 7.1: 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	$O(lgn)$	$O(n)$

7.5 Segment Tree

Segment Tree is a static full binary tree similar to heap that is used for storing the intervals or segments. ‘Static’ here means once the data structure is build, it can not be modified or extended. Segment tree is a data structure that can efficiently answer numerous *dynamic range queries* problems (in logarithmic time) like finding minimum, maximum, sum, greatest common divisor, least common denominator in array. The “dynamic” means there are constantly modifications of the value of elements (not the tree structure). For instance, given a problem to find the index of the minimum/maximum/-sum of all elements in an given range of an array: [i:j].

Definition Consider an array A of size n and a corresponding Segment Tree T (here a range $[0, n-1]$ in A is represented as $A[0:N-1]$):

1. The root of T represents the whole array $A[0:N-1]$.
2. Each internal node in the Segment Tree T represents the interval of $A[i:j]$ where $0 < i < j < n$.
3. Each leaf in T represents a single element $A[i]$, where $0 \leq i < N$.
4. If the parent node is in range $[i, j]$, then we separate this range at the middle position $m = (i + j)/2$, the left child takes range $[i, m]$, and the right child take the interval of $[m+1, j]$.

Because in each step of building the segment tree, the interval is divided into two halves, so the height of the segment tree will be $\log N$. And there will be totally N leaves and $N-1$ number of internal nodes, which makes the total number of nodes in segment tree to be $2N - 1$ and make the segment tree a *full binary tree*.

Here, we use the Range Sum Query (RSQ) problem to demonstrate how segment tree works:

7.2 307. 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.

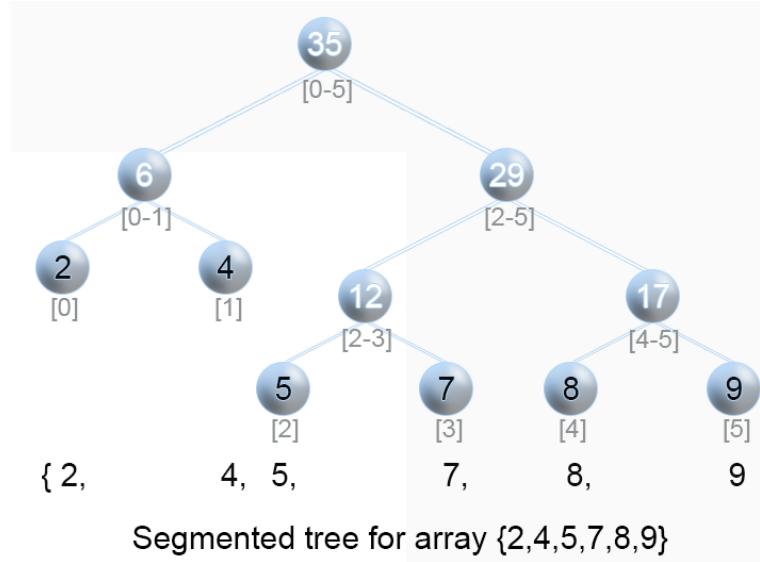


Figure 7.5: Illustration of Segment Tree.

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

```

```

4     if s == e:
5         return self._TreeNode(nums[s])
6
7     m = (s + e)//2
8     # divide
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 7.5, 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.
14                    n-1)
15
16            def update(self, i, val):
17                self._updateNode(i, val, self.st, 0, self.n -1)
18
19            def sumRange(self, i, j):
20                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 $O(\log n)$.

7.6 Trie for String

Definition Trie comes from the word reTrieval. In computer science, a trie, also called digital tree, radix tree or prefix tree which like BST is also a kind of search tree for finding substring in a text. We can solve string matching in $O(|T|)$ time, where $|T|$ is the size of our text. This purely algorithmic approach has been studied extensively in the algorithms: Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp. However, we entertain the possibility that multiple queries will be made to the same text. This motivates the development of data structures that preprocess the text to allow for more efficient queries. Such efficient data structure is Trie, which can do each query in $O(P)$, where P is the length of the pattern string. Trie is an ordered tree structure, which is used mostly for storing strings (like words in dictionary) in a compact way.

1. In a Trie, each child branch is labeled with letters in the alphabet Σ . Actually, it is not necessary to store the letter as the key, because if we order the child branches of every node alphabetically from left to right, the position in the tree defines the key which it is associated to.
2. The root node in a Trie represents an empty string.

Now, we define a trie Node: first it would have a bool variable to denote if it is the end of the word and a children which is a list of 26 children TrieNodes.

```

1 class TrieNode:
2     # Trie node class
3     def __init__(self):
4         self.children = [None]*26
5         # isEndOfWord is True if node represent the end of the
6         # word
6         self.isEndOfWord = False

```

Compressed Trie

- Compress unary nodes, label edges by strings

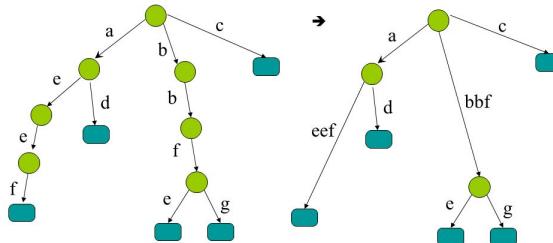


Figure 7.6: Trie VS Compact Trie

Compact Trie If we assign only one letter per edge, we are not taking full advantage of the trie's tree structure. It is more useful to consider compact or compressed tries, tries where we remove the one letter per edge constraint, and contract non-branching paths by concatenating the letters on these paths. In this way, every node branches out, and every node traversed represents a choice between two different words. The compressed trie that corresponds to our example trie is also shown in Figure 7.6.

Operations: INSERT, SEARCH Both for INSERT and SEARCH, it takes $O(m)$, where m is the length of the word/string we want to insert or search in the trie. Here, we use an LeetCode problem as an example

showing how to implement INSERT and SEARCH. Because constructing a trie is a series of INSERT operations which will take $O(n * m)$, n is the total numbers of words/strings, and m is the average length of each item. The space complexity for the non-compact Trie would be $O(N * |\Sigma|)$, where $|\Sigma|$ is the alphlbelical size, and N is the total number of nodes in the trie structure. The upper bound of N is $n * m$.

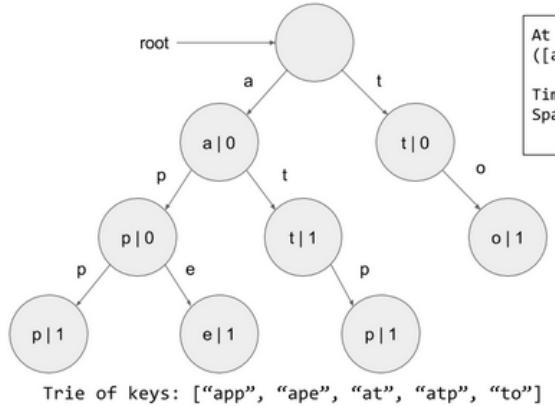


Figure 7.7: Trie Structure

7.1 208. Implement Trie (Prefix Tree) (medium). Implement a trie with insert, search, and startsWith methods.

```

1 Example:
2 Trie trie = new Trie();
3 trie.insert("apple");
4 trie.search("apple");    // returns true
5 trie.search("app");     // returns false
6 trie.startsWith("app"); // returns true
7 trie.insert("app");
8 trie.search("app");     // returns true

```

Note: You may assume that all inputs are consist of lowercase letters a-z. All inputs are guaranteed to be non-empty strings.

INSERT with INSERT operation, we woould be able to insert a given word in the trie, when traversing the trie from the root node which is a `TrieNode`, with each letter in word, if its corresponding node is None, we need to put a node, and continue. At the end, we need to set that node's `endofWord` variable to True. thereafter, we would have a new branch starts from that node constructed. For example, when we first insert “app“ as shown in Fig 7.6, 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            new one
12            node.children[loc] = self.TrieNode()
13        # move to the next node
14        node = node.children[loc]
15    # set the flag to true
16    node.is_word = True

```

SEARCH For SEARCH, like INSERT, we traverse the trie using the letters as pointers to the next branch. There are three cases: 1) for word P, if it doesn't exist, but its prefix does exist, then we return False. 2) If we found a matching for all the letters of P, at the last node, we need to check if it is a leaf node where `is_word` is True. STARTWITH is just slightly different from SEARCH, it does not need to check that and return True after all letters matched.

```

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

```

- 7.1 336. 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", "abeddcba",
       "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
remainder is a palindrome
9         self.pali_indices = set()
10
11    def insert(self, word, i):
12        trie = self
13        for j, ch in enumerate(word):
14            trie = trie.links[ch]

```

```

15         if word[j+1:] and is_palindrome(word[j+1:]):
16             trie.pali_indices.add(i)
17             trie.index = i
18
19
20     def is_palindrome(word):
21         i, j = 0, len(word) - 1
22         while i <= j:
23             if word[i] != word[j]:
24                 return False
25             i += 1
26             j -= 1
27         return True
28
29
30     class Solution:
31         def palindromePairs(self, words):
32             '''Find pairs of palindromes in O(n*k^2) time and O
33             (n*k) space.'''
34             root = Trie()
35             res = []
36             for i, word in enumerate(words):
37                 if not word:
38                     continue
39                 root.insert(word[::-1], i)
40             for i, word in enumerate(words):
41                 if not word:
42                     continue
43                 trie = root
44                 for j, ch in enumerate(word):
45                     if ch not in trie.links:
46                         break
47                     trie = trie.links[ch]
48                     if is_palindrome(word[j+1:]) and trie.index
49                     is not None and trie.index != i:
50                         # if this word completes to a
51                         # palindrome and the prefix is a word, complete it
52                         res.append([i, trie.index])
53                     else:
54                         # this word is a reverse suffix of other
55                         # words, combine with those that complete to a palindrome
56                         for pali_index in trie.pali_indices:
57                             if i != pali_index:
58                                 res.append([i, pali_index])
59             if '' in words:
60                 j = words.index('')
61                 for i, word in enumerate(words):
62                     if i != j and is_palindrome(word):
63                         res.append([i, j])
64                         res.append([j, i])
65
66     return res

```

Solution2: Moreover, there are always more clever ways to solve these problems. Let us look at a clever way: abcd, the prefix is ". 'a',

'ab', 'abc', 'abcd', if the prefix is a palindrome, so the reverse[abcd], reverse[dc], to find them in the words, the words stored in the words with index is fastest to find. $O(n)$. Note that when considering suffixes, we explicitly leave out the empty string to avoid counting duplicates. That is, if a palindrome can be created by appending an entire other word to the current word, then we will already consider such a palindrome when considering the empty string as prefix for the other word.

```

1  class Solution(object):
2      def palindromePairs(self, words):
3          # 0 means the word is not reversed, 1 means the
4          # word is reversed
5          words, length, result = sorted([(w, 0, i, len(w))
6                                           for i, w in enumerate(words)] +
7                                           [(w[::-1], 1, i, len(w))
8                                           for i, w in enumerate(words)]), len(words) * 2, []
9
10         #after the sorting ,the same string were nearby , one
11         #is 0 and one is 1
12         for i, (word1, rev1, ind1, len1) in enumerate(words
13             ):
14             for j in xrange(i + 1, length):
15                 word2, rev2, ind2, _ = words[j]
16                 #print word1, word2
17                 if word2.startswith(word1): # word2 might
18                     be longer
19                     if ind1 != ind2 and rev1 ^ rev2: # one
20                         is reversed one is not
21                         rest = word2[len1:]
22                         if rest == rest[::-1]: result += (([ind1, ind2],)
23                         if rev2 else ([ind2, ind1],)) # if rev2 is
24                         reversed , the from ind1 to ind2
25                         else:
26                             break # from the point of view , break
27                         is powerful , this way , we only deal with possible
28                         reversed ,
29             return result
30
31

```

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.

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.

8.1 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 subtree rooted at x , items on all of its children subtrees of x are smaller than x . Normally, heap is based on binary tree, which makes it a binary heap. Fig. 8.1 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. 8.1). According to such assigning rule, the node in the tree is mapped and saved in the array by the assigned

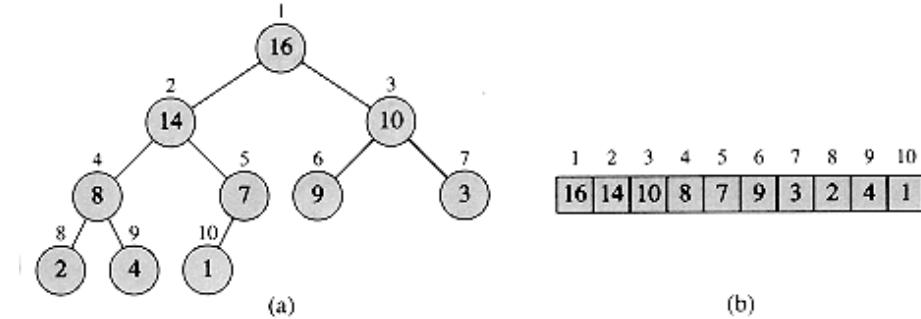


Figure 8.1: Max-heap be visualized with binary tree structure on the left, and implemnted with Array on the right.

index (shown in Fig. 8.1). In heap, we can traverse the imaginary binary 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.

8.1.1 Basic Implementation

The basic method that a heap supports are **insert** and **pop**. Additionally, given an array, to convert it to a heap, we need operation called **heapify**.

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

```

Insert with Floating When we insert an item, we put it at the end of the heap (array) first and increase the size by one. After this, we need to traverse from the last node leaf-to-root, and compare each leaf and parent pair to decide if a swap operation is needed to force the 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             if self.heap[index] < self.heap[p_index]:
5                 # swap
6                 self.heap[index], self.heap[p_index] = self.heap
7                     [p_index], self.heap[index]
8             index = p_index # move up the node
9     def insert(self, val):
10        self.heap.append(val)
11        self.size += 1
12        self.__float__(index = self.size)

```

Pop with Sinking When we pop an item, we first save the root node's value in order to get either the maximum or minimum item in the heap. Then we simply use the last item to fill in the root node. After this, we need to traverse from the root node root-to-leaf, and compare each parent with left and right child. In a min-heap, if the parent is larger than its smaller child node, we swap the parent with the smaller child. This process is called sinking. Same as the insert, $O(\log n)$.

```

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 = li if self.heap[li] < self.heap[ri] else ri
6             if self.heap[index] > self.heap[mi]:
7                 # swap index with mi
8                 self.heap[index], self.heap[mi] = self.heap[mi],
9                     self.heap[index]
10                index = mi
11    def pop(self):
12        val = self.heap[1]
13        self.heap[1] = self.heap.pop()
14        self.size -= 1
15        self.__sink__(index = 1)
16        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 Floating Heapify is a procedure that convert a list to a heap data structure. Similar to the operation insertion, it uses floating. Differently, if we iterating through the list and use insertion operation, each time we try to float, the previous list is already a heap. However, given an unordered array, we can only get the minimum/maximum node floating starting from the bottom. Therefore, we call floating process for the elements in the list in reverse order as a bottom-up manner. The upper bound time complexity is $O(n \log n)$ because we have n call of the float which has an upper bound $O(\log n)$. However, we can prove it actually has a tighter bound by observing that the running time depends on the height of the tree. The proving process is shown on geeksforgeeks¹.

```

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

¹<https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/>

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.

8.1.2 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 8.1. *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 8.1: 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>heapreplace(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.

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]
```

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 hidden 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.

8.2 Priority Queue

A priority queue is 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 use `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()`. 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]

```

Implement with PriorityQueue class Class PriorityQueue() is the same as Queue(), LifoQueue(), they have same member functions as shown in Table 6.9. Therefore, we skip the semantic introduction. PriorityQueue() normally thinks that the smaller the value is the higher the priority is, such as the following example:

```

1 import queue #Python3 needs to use queue, others Queue
2 q = queue.PriorityQueue()
3 q.put(3)
4 q.put(10)
5 q.put(1)
6
7 while not q.empty():
8     next_job = q.get()
9     print('Processing job:', next_job)

```

The output is:

```

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

```

We can also pass by any data type that supports '<' comparison operator, such as tuple, it will use the first item in the tuple as the key.

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

```



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 done 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         return ans
```

8.3 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.

8.4 LeetCode Problems

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

Complete Search

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. **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 algirithms 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 than, 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 algirithms implemented with recursion, such as Depth-first-search for graph, or DFS based tree traversal, or backtracking.

To follow the same organization of Part III, this part is composed of linear search or non-linear search. For each algorithm, we will explain how to use it on cases like: explicit or virtual search space.

- Linear Search (Chapter 9) which describes the common algorithms that carries on Linear data structures: Array, Linked List and String.
- Non-linear Search (Chapter 10) 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 11) 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.

9

Linear Data Structures-based Search

Array Search is to find a **sub-structure** on a given linear data structure(Chapter 6) 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 9.2) 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 9.3). 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 ??).

9.1 Linear Search

As the naive and baseline approach compared with other searching alrogithms, 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).

9.2 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.

9.2.1 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 ,

value > target. Therefore, we simply change the condition of if value < target to if value <= target, then we move to the right side.

```

1 #bisect_right: separate the list into two halves: 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 9.1. However, only two are most commonly used: bisect_left and bisect_right. Let's see some exam-

Table 9.1: 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 insort_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

```

9.2.2 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.

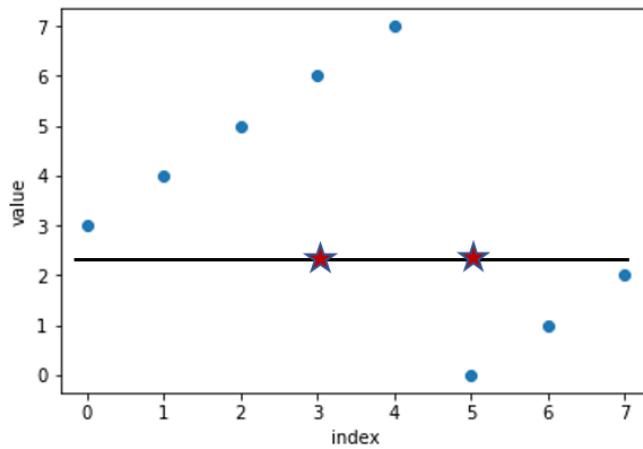


Figure 9.1: Example of Rotated Sorted Array

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              sorted
18              if nums[mid] <= target <= nums[r]:
19                  l = mid+1
20              else:
21                  r = mid-1
22          else:
23              l = mid + 1
24      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++

```

9.2.3 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 $lst[mid] < target$
3. decide which binary search we use: standard, bisect_left/ bisect_right or its mutant.

For example:

9.1 441. 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()

```

9.2 278. 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

```

9.2.4 LeetCode Problems

9.1 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 `len(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 `nums[0]` or larger or equal to `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 -1

```

Standard binary search

1. 611. Valid Triangle Number (medium)
2. 704. Binary Search (easy)
3. 74. Search a 2D Matrix) Write an efficient algorithm that searches for a value in an $m \times n$ matrix. This matrix has the following properties:
 - (a) Integers in each row are sorted from left to right.
 - (b) The first integer of each row is greater than the last integer of the previous row.

For example,
Consider the following matrix:

```
[
    [1, 3, 5, 7],
    [10, 11, 16, 20],
    [23, 30, 34, 50]
]
```

Given target = 3, return true.

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 the 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

```

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)

9.3 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. 9.2, 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 ??).

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

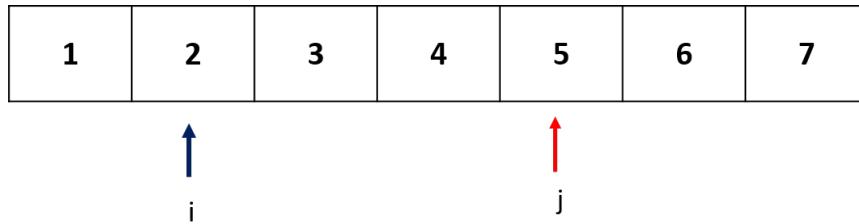


Figure 9.2: Two pointer Example

the median and Floyd's fast-slow pointer algorithm for loop detection in an array/linked list and two pointers to get two sum.

9.3.1 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])

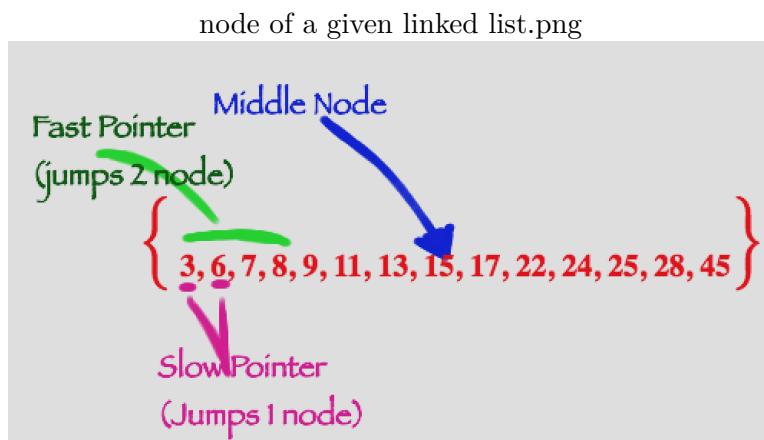


Figure 9.3: Slow-fast pointer to find middle

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. 9.4. 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.

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 traveled 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

```

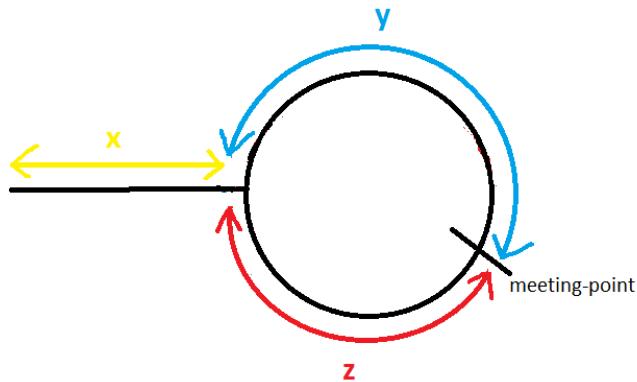


Figure 9.4: Floyd's Cycle finding Algorithm

```

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

```

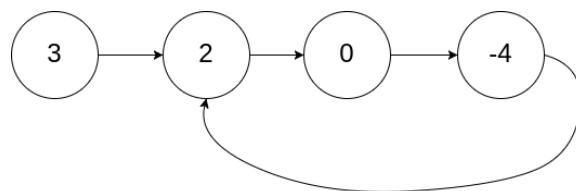


Figure 9.5: One example to remove cycle

In order to remove the cycle as shown in Fig. 9.5, 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

```

9.3.2 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 $[s, s+1, \dots, e-1, e]$, 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 []

```

9.3.3 Sliding Window Algorithm



Figure 9.6: Sliding Window Algorithm

Given an array, imagine that we have a fixed size window as shown in Fig. 9.6, 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 that just slid 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.

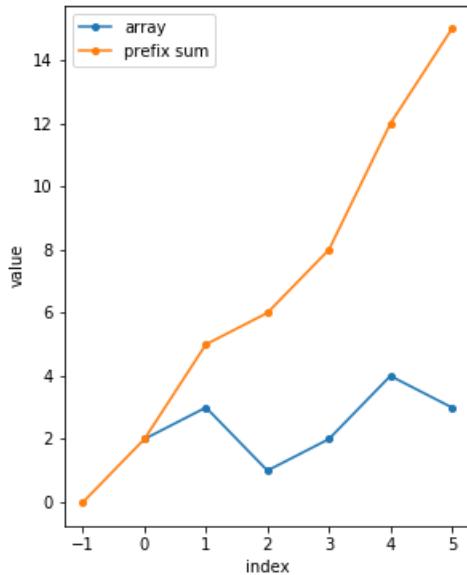


Figure 9.7: The array and the prefix sum

As we have shown in Fig. 9.7, 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 optimal 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.

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 $s[i:j]$ and $s[i-1:j-1]$ should be gained from $s[i-1:j-1]$. 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 ang 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
35
36             # Move the left pointer ahead,
37             i += 1
38
39             # Keep expanding the window
40             j += 1
41             # if bPrint:
42             #     print('move to:', s[i:j+1], i, j)
43         return "" if ans[0] == float("inf") else s[ans[1] : ans[1] + 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.

9.1 930. 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: $A.length \leq 30000$, $0 \leq S \leq A.length$, $A[i]$ 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_{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, 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
location of the i_lo
16            i_hi = i_lo
17            while i_hi < j and sum_window == S and not A[
i_hi]:
18                i_hi += 1
19                if sum_window == S:
20                    ans += i_hi - i_lo + 1
21
22            j += 1 #increase the pointer at last so that we
do not need to check if j<len again
23
24
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;

9.3.4 LeetCode Problems

Sliding Window

9.1 76. Minimum Window Substring

9.2 438. Find All Anagrams in a String

9.3 30. Substring with Concatenation of All Words

9.4 159. Longest Substring with At Most Two Distinct Characters

9.5 567. Permutation in String

9.6 340. Longest Substring with At Most K Distinct Characters

9.7 424. Longest Repeating Character Replacement

10

Elementary Graph-based Search

In this chapter, we cover the comprehensive and basic graph/tree search algorithms. For the graph traversal, breadth-first-search (BFS) and the depth-first-search (DFS) lays the foundation of graph traversal.

10.1 Graph Traversal

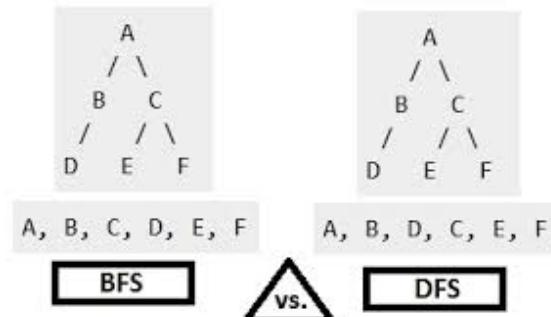


Figure 10.1: BFS VS DFS

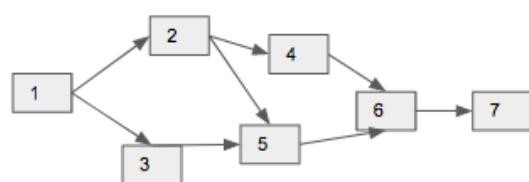


Figure 10.2: Example Graph

The breadth first search (BFS) and the depth first search (DFS) are the two algorithms used for traversing and searching a node in a graph or a tree. They can also be used to find out whether a node is reachable from a given node or not. In Fig. 10.1 shows the BFS and DFS traverse ordering. Starting from a given vertex u , BFS will traverse all of its adjacency nodes v_i and print out them, and then continue to the adjacency nodes of v_i , while the DFS will traverse all of its adjacency nodes, but in a recursively way, which it recursively traverse the adjacency nodes of the current node until reaching to a node that has no outgoing nodes. The implementation of BFS and DFS will be detailed in Section 10.1.1 and 10.1.3. Both BFS and DFS has a time complexity of $O(V + E)$ with adjacency list and $O(V^2)$ with adjacency matrix.

Visiting States Before we move to the implementation, we first define three possible states to mark the visiting status of each node:

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, in DFS, it means returned from the recursive call.
3. GRAY: (0), this is an intermediate state between WHITE and BLACK, which is ongoing, visited but not completed.

A basic DFS and BFS implementation will only need state 1) and 3). While in some advanced extension of DFS and BFS, state 2) might be needed, e.g.

Searching is an universal approach in problem solving. With searching, it literally search in the solution space and find the solutions. In the following section, we will implement BFS and DFS, with example shown in Fig. 10.2 and with the following network structure in Python code:

```

1 adjacency_matrix = {1: [2, 3], 2: [4, 5],
2                         3: [5], 4: [6], 5: [6],
3                         6: [7], 7: []}

```

10.1.1 Breadth-First-Search (BFS)

The aim of BFS algorithm is to traverse the graph as close as possible to the root node. This means we need to traverse the graph level by level. Breadth first search expands nodes in order of their distance from the root. We start from the root, visit the root, and then visit all of root's children and make them as visited. Then, we treat all of these visited children as root, and continue. There are two ways to implement BFS, one is called *Node by Node BFS*, it uses a queue, and deal the elements in the queue as

First In First Out order one by one; the other is called *Level by Level BFS*, in this method, it saves all same-level vertices and visit them all at once.

Node by Node BFS Using a Queue here fits to save the next "root" nodes. The Python implementation is given:

```

1 def bfs_iterative(graph, start):
2     queue, path = [start], []
3     visited = set([start])
4
5     while queue:
6         print(queue)
7         vertex = queue.pop(0) #FIFO
8         path.append(vertex)
9         for neighbor in graph[vertex]:
10            if not neighbor in visited:
11                queue.append(neighbor)
12                visited.add(neighbor)
13    return path
14 print(bfs_iterative(adjacency_matrix, 1))
15 #[1, 2, 3, 4, 5, 6, 7]

```

The changing process of queue is as follows:

```

init [1]
pop out 1, push 1's neighbors 2, 3 in, [2, 3],
pop out 2, push 4, 5 in, [3, 4, 5]
pop out 3, 4, 5 is already in, thus no push, [4, 5]
pop out 4, push 6 in, [5, 6]
pop out 5, [6]
pop out 6, push 7 in, [7]

```

The path is the popped out elements: 1, 2, 3, 4, 5, 6, 7

Level by Level BFS There is another way to implement BFS which is widely used in practice. We find the nodes level by level:

```

1 #implement using a queue
2 def BFSLevel(root):
3     q = [root]
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
13 # print(bfs_iterative(adjacency_matrix, 1))
14 #[1, 2, 3, 4, 5, 6, 7]

```

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

```

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 pathes from src to dst.

10.1.2 Bidirectional Search: Two-end BFS

Definition In normal graph search using BFS/DFS we begin our search in one direction usually from source vertex s toward the goal vertex t , but what if we start search form both direction simultaneously. Bidirectional search is a graph search algorithm which find *smallest path* from source to goal vertex. We just learned that Breadth-first-search suits well for shortest path problem. Because in Level-by-level BFS, it controls the visiting order of nodes by its order to the starting vertex. Therefore, Bidirectional search runs *two simultaneous level-by-level BFS searches* which eventually “meet in the middle” (when two searches intersect) and terminates.

1. Forward search starts form source/initial vertex s toward goal vertex t .
2. Backward search form goal/target vertex t toward source vertex s

Time and Space Complexity Suppose if branching factor of tree is b and distance of goal vertex from source is h , then the normal BFS/DFS searching complexity would be $O(b^h)$. On the other hand, if we execute two search operation then the complexity would be $O(b^{h/2})$ for each search and total complexity would be $O(b^{h/2} + b^{h/2})$ which is far less than $O(b^h)$. Therefore, in many cases bidirectional search is way faster and dramatically reduce the amount of required exploration. Because we need to save nodes at each level in the queue, the maximum nodes we get at the middle $h/2$ will be b^h , this makes the space complexity to $O(b^h)$.

When and How When Bidirectional search can find the shortest path successfully if all the paths are assigned uniform costs.

How When the graph from each side is not balanced: each node has various branch factors. We take two queues: sourceQueue for BFS in forward direction from source to target and targetQueue which is used to do the BFS from the target towards the source in backward direction. We try to alternate between the two queues: sourceQueue and targetQueue; basically in every iteration we choose the smaller queue for the next iteration for the processing which effectively helps in alternating between the two queues only when the swapping between the two queues is profitable.

This helps in the following way: As we go deeper into a graph the number of edges can grow exponentially. Since we are approaching in a balanced way, selecting the queue which has smaller number of nodes for the next iteration, we are avoiding processing a large number of edges and nodes by trying to having the intersection point somewhere in the middle.

Since we are processing both the target and source queue we are not going to much depth from any direction, either in forward direction (i.e, while processing source queue) or in backward direction (i.e, target queue which searches from target to source in backward manner).

Implementation

10.1.3 Depth-First-Search (DFS)

The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Because we need to avoid traverse a cycle which will make the programming run nonstop, we use a data structure, either a hashmap SET or an array of boolean to track the visited nodes.

Recursive DFS Due to the definition of DFS, the recursive implementation is trivial:

```

1 def dfs_recursive(graph, vertex, path=[]):
2     path += [vertex]
3
4     for neighbor in graph[vertex]:
5         if neighbor not in path:
6             path = dfs_recursive(graph, neighbor, path)
7
8     return path
9
10
11 adjacency_matrix = {1: [2, 3], 2: [4, 5],
12                      3: [5], 4: [6], 5: [6],
13                      6: [7], 7: []}
14
15 print(dfs_recursive(adjacency_matrix, 1))

```

```
16 # [1, 2, 4, 6, 7, 5, 3]
```

In the code snippet, line 5 is to check if the current neighbor is visited or not. We can either use a SET or a list of Booleans, or if we know the total vertices are within 32 or 64, we can use bit as shown in Section 15.5.

Iterative DFS In the BFS, we use a Queue to save the nodes in each level. If we use the same way to put the nodes in a data structure, the Stack is used to implement the iterative version of depth first search. Let's see how depth first search works with respect to the following graph:

```
1 def dfs_iterative(graph, start):
2     stack, path = [start], []
3
4     while stack:
5         vertex = stack.pop()
6
7         path.append(vertex)
8         for neighbor in graph[vertex]:
9             if neighbor in path:
10                 continue
11             stack.append(neighbor)
12
13     return path
14
15
16 adjacency_matrix = {1: [2, 3], 2: [4, 5],
17                      3: [5], 4: [6], 5: [6],
18                      6: [7], 7: []}
19
20 print(dfs_iterative(adjacency_matrix, 1))
21 # [1, 3, 5, 6, 7, 2, 4]
```

The process of the stack is:

```
init: [1]
pop out 1, push 2, 3 in, [2, 3]
pop out 3, push 5 in, [2, 5]
pop out 5, push 6 in, [2, 6]
pop out 6, push 7 in, [2, 7]
pop out 7, [2]
pop out 2, push 4 in, [4]
pop out 4
```

The pop out order is the dfs traverse order.

However, from the previous recursive and the iterative implementation, we can see the dfs traverse order is different, this is due to in the iterative, the order of the neighbors of each node is not reversed. If we replace line 8 with the following code, then they will have exactly the same output, which in the tree structure, it can matter.

```
1 for neighbor in graph[vertex][::-1]
```

10.1.4 Backtracking

Backtracking is a variant of Depth-first search, and is a general algorithm for finding all (or some) solutions to some computational problems, that *incrementally* builds candidates to the solutions. 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 that level’s state so that the search process can continue to explore the next choice. Backtracking is all about choices and consequences, this is why backtracking is the most common algorithm for solving *constraint satisfaction problem (CSP)*¹, such as Eight Queens puzzle, Map Coloring problem, Sudoku, Crosswords, and many other logic puzzles. (N-Queens : permutations with backtracking Soduko : counting with backtracking Scheduling: subsets with backtracking <https://www.cs.princeton.edu/~rs/AlgsDS07/24CombinatorialSearch.pdf>)

Backtracking VS DFS The implementation of Backtracking is equivalent to a DFS without restriction that each node can only be visited only and exactly once. In Fig. 10.2, the DFS path is 1->3->5->6->7 (backtrack to 1)->2->4 (backtrack to 2). The only difference is in the *backtracking*, we are not mandatorily restricted to visit each node only and exactly once like the DFS needs which keeps a visited array to track the status. In backtracking, we can have such path 1->3->5->6->7, backtrack to 2, another path 1->2->4->6->7. Therefore, we can find two paths from vertex 1 to vertex 7.

Backtracking VS Exhaustive Search Backtracking helps in solving an overall problem by incrementally builds candidates, 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**.

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

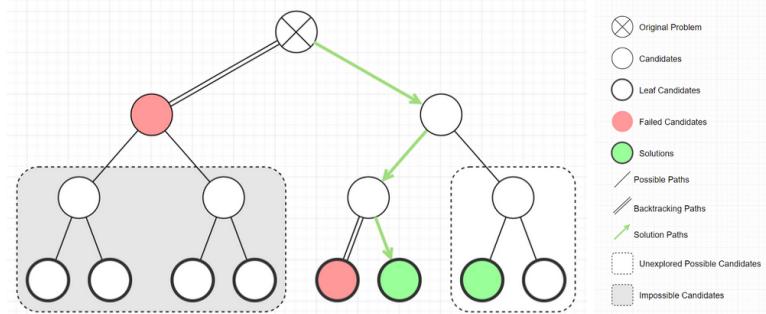


Figure 10.3: Tree of possibilities for a typical backtracking algorithm

Visualize Backtracking A backtracking algorithm can be thought of as a *tree of possibilities*. In this tree, the root node is our starting point, we traverse the possible choices for the next step (all the children nodes). If we reach to end condition (leaf candidates) we succeed and the DFS 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. 10.3.

Application Example Three classical computational problems: combinations and permutations (See in Fig 10.4) (Section 10.1.4 and Section 10.1.4), and puzzles or sudoku problems with constraints (Section 10.1.4) can be solved using backtracking. We should keep in mind that backtracking can help implementing combination and permutation problems, however, they are not the only way to solve these problems.

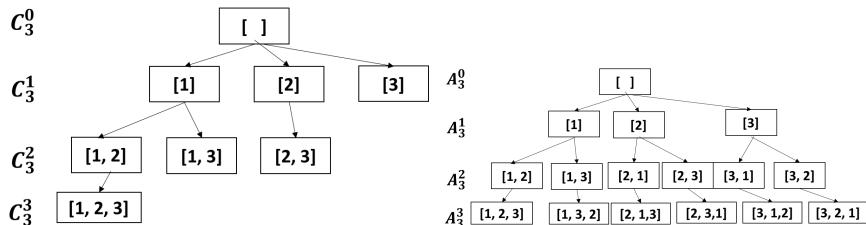


Figure 10.4: The state transfer graph of Combination and Permutation

Combination

Combinations refer 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 $[] \rightarrow [1] \rightarrow [1, 2] \rightarrow [1, 2, 3]$, 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).

The Python implementation is given as below: here d denotes the degree or which level we are currently at. And k controls the return level. s denotes the start position for the current available; $curr$ is used to save the current state or say subset; ans saves all the result and it is a global variable. $curr.pop()$ is the soul for showing it is a backtracking algorithm!

```

1 def combination(d, k, s, curr, ans, nums): #d controls the
2     degree (depth), k is controls the return level, s is the
3     start position for available numbers, curr saves the current
4     result , ans is all the result
5     # collect result
6     ans.append(curr)
7     # END condition
8     if d == k:
9         return
10    # backtracking
11    for i in range(s, len(nums)):
12        curr.append(nums[i])
13        combination(d+1, k, i+1, curr[:], ans, nums) # i+1
14        # because no repeat, make sure use deep copy curr [:]
15        curr.pop()
16
17 nums = [1, 2, 3]
18 ans = []
19 # get a power set: set k = len(nums)
20 combination(0, 3, 0, [], ans, nums)
21 print(ans)
22 #output
23 #[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
```

From the above result we can see it is a DFS traversal. What if we just want to gain result of a single layer at C_n^k (as shown in the Fig. 10.4)? The change is trivial; we can set the return level k to the level we want and collect the result only at the end condition.

```
1 # replace the 2nd–6th line of code above with the following
```

```

2 if d == k:
3     ans.append(curr)
4     return

```

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

Time Complexity

Permutation

Permutations refer to the permutation of n things taken k at a time without repetition, the math formula is A_n^k . Compared with combination, [a, b] and [b, a] would be considered as different solution. The relation of the number of combination and permutation solution can be described in formula: $C_n^k = \frac{A_n^k}{k!}$, where $k! = k * (k - 1) \dots * 1$. So $A \geq C$.

Python Template

```

1 # template for Ank, d==k, n==
2 def permute(nums, n, k):
3     """
4     :type n: int
5     :type k: int
6     :rtype: List[List[int]]
7     """
8     ans = []
9     def A_n_k(d, k, used, curr): #d controls the degree (depth),
10        k is controls the depth of the state transfer
11        nonlocal ans
12        if d==k:
13            ans.append(curr)
14            return
15        for i in range(n):
16            if used[i]:
17                continue
18            used[i] = True
19            curr.append(nums[i])
20            A_n_k(d+1, k, used[:], i+1, curr[:])
21            curr.pop()
22            used[i] = False
23    A_n_k(0, k, [False]*n, 0, [])
24    return ans

```

There is another way with brute force, each time we have the option choose this element or not to choose this element. Include this content.

To see the application of this, to go chapter array with the subset

Puzzles and Sudoku

We will now create a Sudoku solver using backtracking by encoding our problem, goal and constraints in a step-by-step algorithm. Problem

```
In [34]: permute([1,2,3,4],
```

```
Out[34]: [[1, 2],
           [1, 3],
           [1, 4],
           [2, 1],
```

Given a, possibly, partially filled grid of size ‘n’, completely fill the grid with number between 1 and ‘n’. Goal is defined for verifying the solution. Once the goal is reached, searching terminates. A fully filled grid is a solution if:

1. Each row has all numbers form 1 to ‘n’.
2. Each column has all numbers form 1 to ‘n’.
3. Each sub-grid (if any) has all numbers form 1 to ‘n’.

Constraints are defined for verifying each candidate. A candidate is valid if:

1. Each row has unique numbers form 1 to ‘n’ or empty spaces.
2. Each column has unique numbers form 1 to ‘n’ or empty spaces.
3. Each sub-grid (if any) has unique numbers form 1 to ‘n’ or empty spaces.

Termination conditions

Typically, backtracking algorithms have termination conditions other than reaching goal. These help with failures in solving the problem and special cases of the problem itself.

1. There are no empty spots left to fill and the candidate still doesn’t qualify as a the solution.
2. There are no empty spots to begin with, i.e., the grid is already fully filled.

Step-by-step algorithm

Here’s how our code will “guess” at each step, all the way to the final solution:

1. Make a list of all the empty spots.

2. Select a spot and place a number, between 1 and ‘n’, in it and validate the candidate grid.
3. If any of the constraints fails, abandon candidate and repeat step 2 with the next number. Otherwise, check if the goal is reached.
4. If a solution is found, stop searching. Otherwise, repeat steps 2 to 4.

10.1 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:

1. Each of the digits 1-9 must occur exactly once in each row.
2. Each of the digits 1-9 must occur exactly once in each column.
3. 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 ‘?’.

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 10.5: Example sudoku puzzle and its solution

Solution: backtrack and bits to record states. The idea is very simple: we use three vectors with length 9 to record the state of each row, col and grid of 3 by 3. The value of the vector is int, which is initialized as 0. If 5 appears in the row, we change that row[index]’s int’s fifth bit into 1.

The main algorithm is complete search (linear) with backtrack. We have a helper function that takes (i,j) as input. We move i,j in the order from left to right and up to down. We grid that has ‘?’ we simply try to find a value from the range of [1,9], and also not appear in any of the recorded states. And we recursively call helper(next_i, next_j),

if it returns False, we backtrack to the previous state. If we can not find a possible value to even try, we return False directly.

```

1 def solveSudoku(self, board):
2     """
3         :type board: List[List[str]]
4         :rtype: void Do not return anything, modify board in-
5             place instead.
6     """
7     row_state = [0]*9
8     col_state = [0]*9
9     grid_state = [0]*9
10    rows = len(board)
11    cols = len(board[0])
12
13    empty_spots = []
14    # initialize the state
15    for i in range(rows):
16        for j in range(cols):
17            if board[i][j] != '.':
18                # set that bit to 1
19                row_state[i] |= 1 << (int(board[i][j])-1)
20                col_state[j] |= 1 << (int(board[i][j])-1)
21                grid_index = (i//3)*3 + (j//3)
22                grid_state[grid_index] |= 1 << (int(board[i][j])-1)
23            else:
24                empty_spots.append((i, j))
25
26    n = len(empty_spots)
27
28    def helper(index):
29        if index == n:
30            return True
31        i, j = empty_spots[index]
32
33        for v in range(9):
34            row_bit = (1 << v) & row_state[i] != 0
35            col_bit = (1 << v) & col_state[j] != 0
36            grid_index = (i//3)*3 + (j//3)
37            grid_bit = (1 << v) & grid_state[grid_index]
38            if not row_bit and not col_bit and not grid_bit:
39                board[i][j] = str(v+1)
40                # change state
41                # set that bit to 1
42                row_state[i] |= 1 << (v)
43                col_state[j] |= 1 << (v)
44                grid_state[grid_index] |= 1 << (v)
45                if helper(index+1)==True:
46                    return True
47

```

```

48     #backtrack , and clear states
49     board[i][j] = '.'
50     row_state[i] &= ~(1 << (v))
51     col_state[j] &= ~(1 << (v))
52     grid_state[grid_index] &= ~(1 << (v))
53     # no possible digit to try
54     return False
55
56     helper(0)

```

10.1.5 Summary

BFS and DFS are two of the most universal algorithms for solving practical problems. Each suits better than the other to specific type of problems.

- For BFS, it suits problems that ask the shortest paths(unweighted) from a certain source node to a certain destination, whether it is single sourced or all-pairs. Or in some cases, the questions requires us only traverse the graph for a certain steps (levels). Because BFS is iterative and can traverse the nodes level by level.
- For DFS, it is better for the weighted optimization problem, that we are required to count all possible paths or to get the best out of all possible paths. Because DFS has the advantage of saving the result of overlapping subproblem to avoid extra computation.
- Use either BFS or DFS when we just need to check correctness (whether we can reach a given state to another).

10.2 Tree Traversal

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees. In all, we have depth first traversal and breath first traversal. For the Depth First Traversals, we have inorder, preorder and postorder traversal, with the following example shown in Fig. 10.6.

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

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

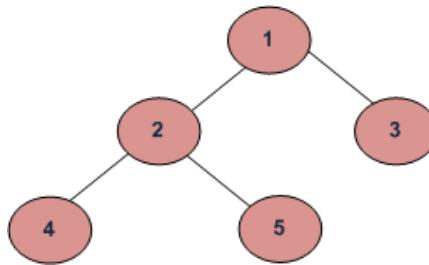


Figure 10.6: Binary Tree

10.2.1 Depth-First-Search Tree Traversal Implementation

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

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 PostTraversal:

```

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 treveral's iterative implmentation 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

```

10.2.2 Level Order Tree Traversal Implementation

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]
```

10.2.3 Time complexity of Binary Tree

If we spent $O(n)$ to convert $T(n)$ to $2T(n/2)$. We have the following deduction:

$$\begin{aligned}
 T(n) &= 2T(n/2) + O(n) \\
 &= 2 * 2T(n/4) + O(n) + O(n) \\
 &= O(n \log n)
 \end{aligned} \tag{10.1}$$

which is the same as merge sort. If the divide cost is only $O(1)$.

$$\begin{aligned}
 T(n) &= 2T(n/2) + O(1) \\
 &= 2 * 2T(n/4) + O(1) + O(1) \\
 &= n + (1 + 2 + 4 + \dots + n) \\
 &\approx n + 2n \\
 &\approx O(n)
 \end{aligned} \tag{10.2}$$

10.2.4 Exercise

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)

```

10.3 Exercise

10.3.1 Backtracking

77. Combinations

```

1 Given two integers n and k, return all possible combinations of
k numbers out of 1 ... n.
2
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 ]

```

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.

11

Advanced Graph-based Search

This chapter is built upon on the last chapter 10. Basically the advanced application of DFS and BFS.

The first several sections, we talk about the advanced applications of DFS, including Connected Components in Graph (Section 11.1), Topological Sort(), Minimum Spanning Tree (MST) ??.

11.1 Connected Components in Graph

Finding connected components for an undirected graph is an easier task compared in the directed graph. DFS can be used to mark the connected components and count the total number of connected components.

11.1.1 In Undirected Graph

The connected components are defined as Fig 11.1. We simply iterate all vertices in V that is not visited before, and call DFS on these not visited nodes: We simple need to do either BFS or DFS starting from every unvisited vertex, and we get all strongly connected components. Below are steps based on DFS.

```

1) Initialize all vertices as not visited.
2) Do following for every vertex 'v'.
   (a) If 'v' is not visited before, call DFSUtil(v)
   (b) Print new line character
5
6 DFSUtil(v)
7 1) Mark 'v' as visited.
8 2) Print 'v'
9 3) Do following for every adjacent 'u' of 'v'.

```

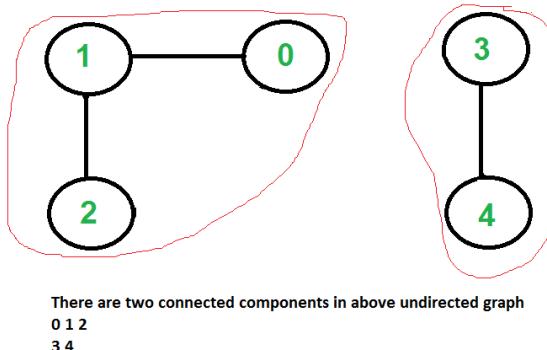


Figure 11.1: The connected components in undirected graph

```
10 If 'u' is not visited , then recursively call DFSUtil(u)
```

11.1 130. 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
14     moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]
15     # find all connected components to the edge 0, and mark
16     # them as -1,
17     # then flip all 0s in the other parts
18     # change the -1 to 0s
19     visited = [[False for c in range(cols)] for r in range(
20             rows)]
21     def dfs(x, y): # (x, y) is the edge 0s
22         for dx, dy in moves:
23             nx = x + dx
24             ny = y + dy
25             if nx < 0 or nx >= rows or ny < 0 or ny >= cols
26                 continue
27             if board[nx][ny] == 'O' and not visited[nx][ny]:
28                 visited[nx][ny] = True
29                 dfs(nx, ny)
30     # first and last col
31     for i in range(rows):
32         if board[i][0] == 'O' and not visited[i][0]:
33             visited[i][0] = True
34             dfs(i, 0)
35         if board[i][-1] == 'O' and not visited[i][-1]:
36             visited[i][-1] = True
37             dfs(i, cols-1)
38     # first and last row
39     for j in range(cols):
40         if board[0][j] == 'O' and not visited[0][j]:
41             visited[0][j] = True
42             dfs(0, j)
43         if board[rows-1][j] == 'O' and not visited[rows-1][
44             j]:
45             visited[rows-1][j] = True
46             dfs(rows-1, j)
47     for i in range(rows):
48         for j in range(cols):
49             if board[i][j] == 'O' and not visited[i][j]:
50                 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    # first and last col
25    for i in range(rows):
26        if board[i][0] == 'O':
27            board[i][0] = '-1'
28            dfs(i, 0)
29        if board[i][-1] == 'O' :
30            board[i][-1] = '-1'
31            dfs(i, cols-1)
32    # # first and last row
33    for j in range(cols):
34        if board[0][j] == 'O':
35            board[0][j] = '-1'
36            dfs(0, j)
37        if board[rows-1][j] == 'O':
38            board[rows-1][j] = '-1'
39            dfs(rows-1, j)
40    for i in range(rows):
41        for j in range(cols):
42            if board[i][j] == 'O':
43                board[i][j] = 'X'
44            elif board[i][j] == '-1':
45                board[i][j] = 'O'
46            else:
47                pass

```

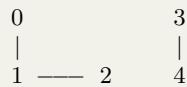
11.2 323. Number of Connected Components in an Undirected Graph (medium).

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:

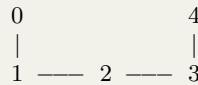
Input: $n = 5$ and edges = $[[0, 1], [1, 2], [3, 4]]$



Output: 2

Example 2:

Input: $n = 5$ and edges = $[[0, 1], [1, 2], [2, 3], [3, 4]]$



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 adjacency list. Because it is undirected, one edge we need to add two directions in the adjacency 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

```

11.1.2 Directed Graph

When the edges are directed, the *strongly connected components (SCC)* are defined as a connected components, where any pair of vertices u and v in the SCC, we can find a path from u to v . There are two popular algorithms to find SCC: Kosaraju's and Tarjans'.

11.2 Cycle Detection

DFS can be extended to detect cycle in $O(V + E)$ time complexity. The detection algorithm differs in the Directed and undirected graph. In the process of DFS, if we encounter an already visited vertex, if in an undirected graph, we detect a cycle. However, in a directed graph, this can possibly because there are two different paths to reach this vertex.

Cycle Detection in Undirected Graph The cycle detection in undirected graph is more straightforward and simpler in implementation compared with in the Directed graph. The rule is: We do DFS traversal of the given graph. For every visited vertex v , if there is an adjacent μ such that μ is already visited and not a parent of v , then there is a cycle.

```

1 from collections import defaultdict
2
3 #This class represents a undirected graph using adjacency list
4 #representation
4 class Graph:
5
6     def __init__(self, vertices):
7         self.V= vertices #No. of vertices
8         self.graph = defaultdict(list) # default dictionary to
9         store graph
10
11     # function to add an edge to graph
12     def addEdge(self,v,w):
13         self.graph[v].append(w) #Add w to v_s list
14         self.graph[w].append(v) #Add v to w_s list
15

```

```

16     # A recursive function that uses visited[] and parent to
17     # detect
18     # cycle in subgraph reachable from vertex v.
19     def isCyclicUtil(self, v, visited, parent):
20
21         #Mark the current node as visited
22         visited[v] = True
23
24         #Recur for all the vertices adjacent to this vertex
25         for i in self.graph[v]:
26             # If the node is not visited then recurse on it
27             if visited[i] == False:
28                 if (self.isCyclicUtil(i, visited, v)):
29                     return True
30             # If an adjacent vertex is visited and not parent of
31             # current vertex,
32             # then there is a cycle
33             elif parent != i:
34                 return True
35
36         return False
37
38     #Returns true if the graph contains a cycle, else false.
39     def isCyclic(self):
40         # Mark all the vertices as not visited
41         visited = [False] * (self.V)
42         # Call the recursive helper function to detect cycle in
43         # different
44         #DFS trees
45         for i in range(self.V):
46             if visited[i] == False: #Don't recur for u if it is
47                 already visited
48                 if (self.isCyclicUtil(i, visited, -1)) == True:
49                     return True
50
51     return False

```

Cycle Detection in Directed Graph Here, we need to use three states, WHITE, BLACK, and GRAY. When we are visiting a GRAY state, it means we are revisiting a visiting node, and the nodes in the cycle, if we don't return there is a cycle, the node in the cycle will never reach to complete visited state.

```

1 # Python program to detect cycle
2 # in a graph
3 from collections import defaultdict
4 WHITE = -1
5 GRAY = 0
6 BLACK = 1
7 class Graph():
8     def __init__(self, vertices):

```

```

9     self.graph = defaultdict(list)
10    self.V = vertices
11
12    def addEdge(self, u, v):
13        self.graph[u].append(v)
14
15    def isCyclicUtil(self, v, visited, stack):
16        stack.append(v)
17        if visited[v] == GRAY:
18            print('re visiting:', v)
19            return True
20
21        visited[v] = GRAY # visiting
22
23        for neighbour in self.graph[v]:
24            if visited[neighbour] != BLACK:
25                if self.isCyclicUtil(neighbour, visited, stack) ==
True:
26                    return True
27        stack.remove(v)
28        visited[v] = BLACK
29        return False
30
31    # Returns true if graph is cyclic else false
32    def isCyclic(self):
33        visited = [WHITE] * self.V
34
35        #recStack = [False] * self.V
36        for node in range(self.V):
37            stack = []
38            if visited[node] != BLACK:# visit it if its not
completed
39            if self.isCyclicUtil(node, visited, stack) == True:
40                print(stack)
41                return True
42        return False
43

```

Here let us see some examples running the above code:

```

1  g = Graph(4)
2  g.addEdge(0, 1)
3  g.addEdge(0, 2)
4  g.addEdge(1, 2)
5  #g.addEdge(2, 0)
6  g.addEdge(2, 3)
7  #g.addEdge(3, 3)
8  if g.isCyclic() == True:
9      print("Graph has a cycle")
10 else:
11     print("Graph has no cycle")
12 # output
13 # Graph has no cycle
14

```

If we put the edge $2 \rightarrow 0$ and $3 \rightarrow 0$, we will get the cycle, plus the nodes in the stack can tell us what is the cycle.

11.3 Topological Sort

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. vs Depth First Traversal (DFS): For example, the topological sort of the Fig 11.2 can

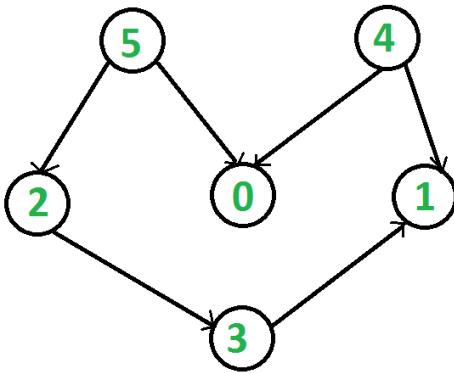


Figure 11.2: Topological sort

be $[5, 2, 3, 4, 0, 1]$ and another topological sorting of the following graph is $[4, 5, 2, 3, 1, 0]$. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges). While the DFS printing of this figure is $[5, 2, 3, 1, 0, 4]$.

Topological sort traverse the graph same way as DFS, In DFS, we print a vertex when its in state GRAY (visited but not complete) and then recursively call DFS for its adjacent vertices. In topological sorting, we only print a vertex when its in state BLACK (visited and completed). And the topological sort is in reverse order such printing, thus using a temporary stack to save the completed vertices. Finally, print contents of stack. Therefore, a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Implementation in Acyclic DAG

```

1 def topologicalSort(self):
2     # Mark all the vertices as not visited
3     visited = [False]*self.V
4     stack = []
5
6     # Sort starting from all vertices one by one
7     for i in range(self.V):
8         if visited[i] == False:
  
```

```

9         self.topologicalSortUtil(i, visited, stack)
10    # Print contents of stack
11    print stack
12
13 # A modified DFS
14 def topologicalSortUtil(self, v, visited, stack):
15     visited[v] = True
16     for i in self.graph[v]:
17         if visited[i] == False:
18             self.topologicalSortUtil(i, visited, stack)
19     # complete visiting v, push current node in stack
20     stack.insert(0,v)

```

11.4 Minimum Spanning Trees

11.4.1 Kruskal and Prim Algorithm

11.5 Single-Source Shortest Paths

11.5.1 The Bellman-Ford Algorithm

11.5.2 Dijkstra's Algorithm

11.6 All-Pairs Shortest Paths

The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

11.6.1 The Floyd-Warshall Algorithm

Part V

Advanced Algorithm Design

This part covers two important and advanced paradigm of the four kingdoms: dynamic programming (Chapter 12) and greedy algorithms (Chapter 13).

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.

12

Dynamic Programming

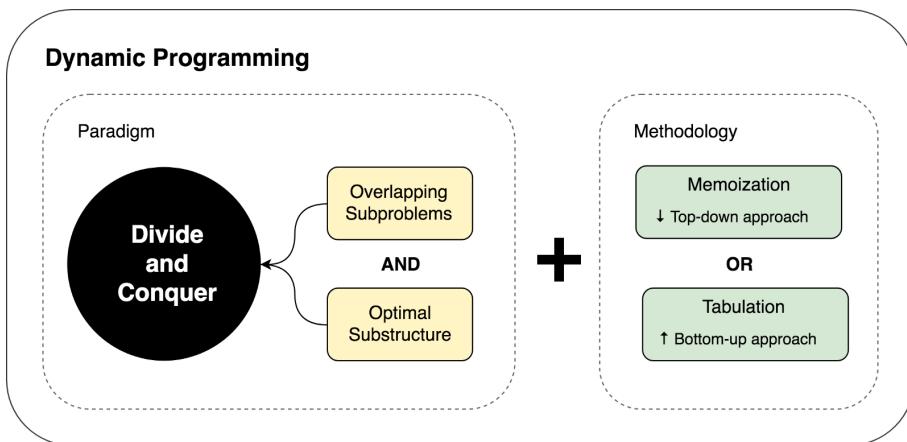


Figure 12.1: Dynamic Programming Chapter Recap

As introduced in Divide-and-Conquer in Chapter 4, dynamic programming is applied on problems that its subproblems shows overlapping feature when divided and solved in Divide-and-Conquer manner. We use the recurrence function: $T(n) = T(n - 1) + T(n - 2) + \dots + T(1) + f(n)$, to show its special character.

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.

Subproblem Graph If we treat each subproblem as a vertex, and the relation between subproblems as edges, we can get a directed subproblem

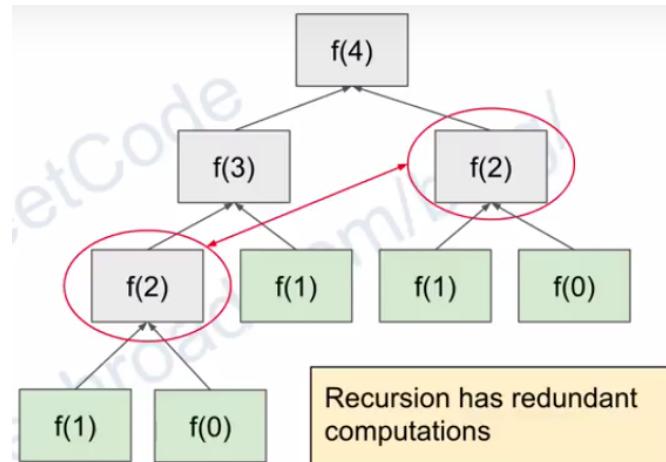
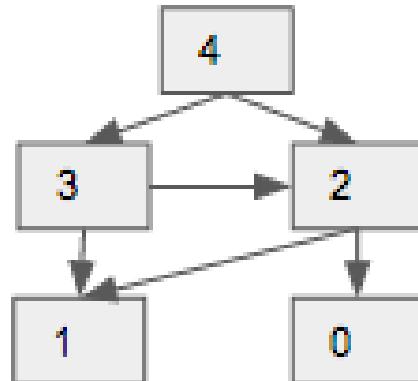


Figure 12.2: Fibonacci number's Recursion Tree

Figure 12.3: Subproblem Graph for Fibononacci Sequence till $n=4$.

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. 12.3 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 12.1. 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 .

12.1 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 12.1 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 2.3.

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 12.1.1) and Longest Increasing Subsequence (subsection 12.1.2) in this section to show us in practice how to do the *memoization* and *tabulation*.

12.1.1 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 12.3, 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]

```

12.1.2 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.

12.1 300. 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?

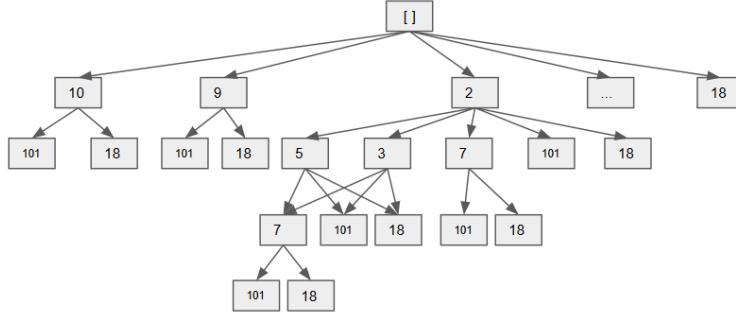


Figure 12.4: State Transfer Tree Structure for LIS, each path represents a possible solution. Each arrow represents a move: find an element in the following elements that's larger than the current node.

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 12.4, 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
be any number in nums
11    #lens[0] = 1
12    while q:
13        new_q = set()
14        for idx in q:
# search for number that is larger than
current
15            for j in range(idx+1, len(nums)):
16                if nums[j] > nums[idx]:
17                    lens[j] = max(lens[j], lens[idx]+1)
18                    new_q.add(j)
19    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. 12.1, $\max(LIS(j) + 1)$ 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 12.5. 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 12.5: The solution to LIS.

$$f(i) = \begin{cases} 1 + \max(f(j)), & 0 < j < i, arr[j] < arr[i]; \\ 1 & \text{otherwise} \end{cases} \quad (12.1)$$

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.

12.2 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 12.2.1) 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 12.2.2) in implementing the dynamic programming solution and to answer the *how* question.
3. Dos and Do nots (Section 12.2.3)
4. Generalization (Section 12.2.4).

12.2.1 Two properties

In order for the dynamic programming to apply, we must characterize two properties: overlapping subproblems and optimal substructure. From Example 12.1, 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.

12.2.2 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.

12.2.3 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.

12.2.4 Generalization: Steps to Solve Dynamic Programming

1. Read the question, using Sec 12.2.3 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 12.2.2 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: 1. It is a natural transformation from normal Complete Search recursion 2. Compute sub-problems only when necessary (sometimes this is faster)	Pro: 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: 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	Cons: 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 12.6: 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 12.6.

12.3 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.

12.3.1 Example of optimizing $O(2^n)$ 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))
7     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))]
5     def divideConquerMemo(x, y):
6         #nonlocal memo
7         if x == len(triangle):
8             return 0
9         if memo[x][y] == maxsize:
10             memo[x][y] = triangle[x][y] + min(divideConquerMemo(
11                 x+1, y), divideConquerMemo(x+1, y+1))
12             return memo[x][y]
13     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]

```

12.3.2 Example of optimizing $O(n^3)$ 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

```

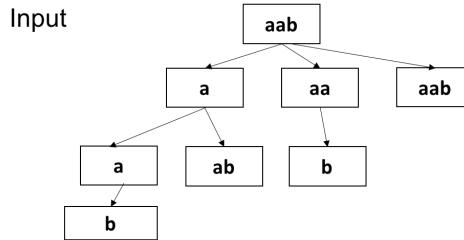


Figure 12.7: State Transfer for the panlindrom splitting

```

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.

12.3.3 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.

12.4 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:

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \quad (12.2)$$

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. 12.2, we have the following inequation:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor lg(\lfloor n/2 \rfloor)) + n \\ &\leq cnlg(n/2) + n \\ &\leq cnlg n - cnlg 2 + n \\ &\leq cnlg n - cn + n \\ &\leq cnlg n, \text{ if } c \geq 1, \end{aligned} \quad (12.3)$$

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:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1, \quad (12.4)$$

A reasonable and logical guess for this recurrence is $T(n) = O(n)$, so that we obtain

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1, \\ &= cn + 1 \end{aligned} \quad (12.5)$$

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.

12.5 Exercises

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 3×3 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, $\text{costs}[0][0]$ is the cost of painting house 0 with color red; $\text{costs}[1][2]$ is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Solution: state: 0, 1, 2 colors $\text{minCost}[i] = \text{till } i$ the mincost for each color for color 0: paint 0 [0] = $\min(\text{minCost}[i-1][1], \text{minCost}[i-1][2]) + \text{costs}[i][0]$

paint 1 [1]

$\text{minCost}[i] = [0,1,2]$, i for i in $[0,1,2]$

$\text{answer} = \min(\text{minCost}[-1])$

```

1 def minCost(self, costs):
2     """
3         :type costs: List[List[int]]
4         :rtype: int
5     """
6     if not costs:
7         return 0
8     if len(costs) == 1:
9
10    index is 0
11    dp = [[0] * 3] * len(costs)
12    dp[0] = costs[0]
13
14    for i in range(1, len(costs)):
15        for j in range(3):
16            if j == 0:
17                dp[i][j] = min(dp[i - 1][1], dp[i - 1][2]) + costs[i][j]
18            if j == 1:
19                dp[i][j] = min(dp[i - 1][0], dp[i - 1][2]) + costs[i][j]
20            if j == 2:
21                dp[i][j] = min(dp[i - 1][0], dp[i - 1][1]) + costs[i][j]
22
23    return min(dp[-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 = "aadbbbaccc"
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     ]);
17         left_max[i] = Math.max(left_max[i - 1],
18         localMax);
19     }
20     //search for left_min
21     for (int i = 1; i < size; i++) {
22         localMin = Math.min(nums[i], localMin + nums[i
23     ]);
24         left_min[i] = Math.min(left_min[i - 1],
25         localMin);
26     }
27
28     right_max[size - 1] = right_min[size - 1] = nums[
29     size - 1];
30     //search for right_max
31     localMax = nums[size - 1];
32     for (int i = size - 2; i >= 0; i--) {
33         localMax = Math.max(nums[i], localMax + nums[i
34     ]);
35         right_max[i] = Math.max(right_max[i + 1],
36         localMax);
37     }
38     //search for right_min
39     localMin = nums[size - 1];
40     for (int i = size - 2; i >= 0; i--) {
41         localMin = Math.min(nums[i], localMin + nums[i
42     ]);
43         right_min[i] = Math.min(right_min[i + 1],
44         localMin);
45     }
46     //search for separate position
47     int diff = 0;
48     for (int i = 0; i < size - 1; i++) {
49         diff = Math.max(Math.abs(left_max[i] -
50             right_min[i + 1]), diff);
51         diff = Math.max(Math.abs(left_min[i] -
52             right_max[i + 1]), diff);
53     }
54     return diff;
55 }
```

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.

$$\min_local[i] = \begin{cases} \min(\min_local[i-1] * \text{nums}[i], \text{nums}[i]), & \text{nums}[i] < 0; \\ \min(\max_local[i-1] * \text{nums}[i], \text{nums}[i]) & \text{otherwise} \end{cases} \quad (12.6)$$

$$\max_local[i] = \begin{cases} \max(\max_local[i-1] * \text{nums}[i], \text{nums}[i]), & \text{nums}[i] > 0; \\ \max(\min_local[i-1] * \text{nums}[i], \text{nums}[i]) & \text{otherwise} \end{cases} \quad (12.7)$$

```

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 (
               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    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 $f[i-A[i]]+V[i]$.

12.6 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 12.8: Summary of different type of dynamic programming problems

13

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.

13.1 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.

13.2 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.

13.2.1 Greedy-choice Property

Greedy

[Subscribe](#) to see which companies asked this question

You have solved **7 / 38** problems.

[Show problem tags](#)

#	Title	Acceptance
✓ 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 13.1: Screenshot of Greedy Catalog, showing the frequency and difficulty of this type in the real coding interview

Part VI

Math and Bit Manipulation

14

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 14.1 - 14.4 focuses on numerical sorting and selection algorithms. In addition, Section 14.5 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 14.6, 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 14.1): Bubble Sort, Insertion Sort, Selection Sort;
- $O(n \log n)$ (Section 14.2) Sorting: merge sort, quick sort, and Quick Select;

- $O(n + k)$ Sorting (Section 14.3): Counting Sort, where k is the range of the very first and last key.
- $O(n + k)$ Sorting (Section 14.4): Bucket Sort and Radix Sort.
- Lexicographical Order (Section 14.5): In this section, the algorithms that do lexicographical order is
- Python Built-in Sort (Section 14.6):

14.1 $O(n^2)$ 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.

14.1.1 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. 14.1.

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.

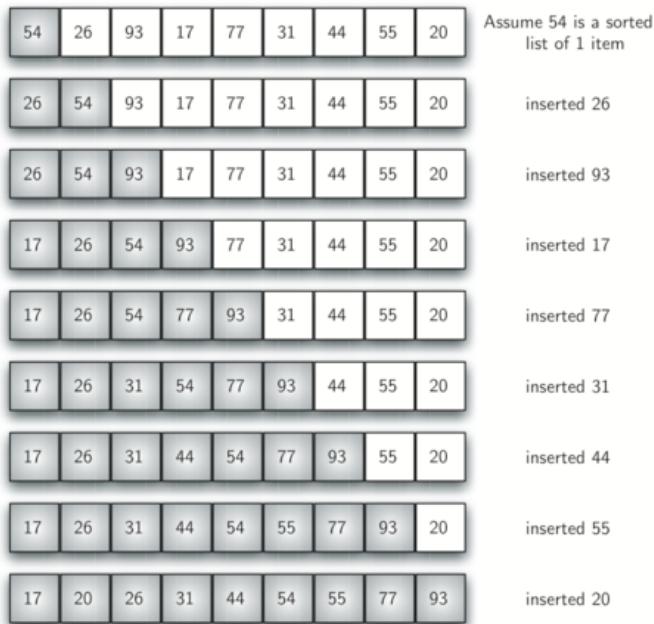


Figure 14.1: The whole process for insertion sort

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

```

14.1.2 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.

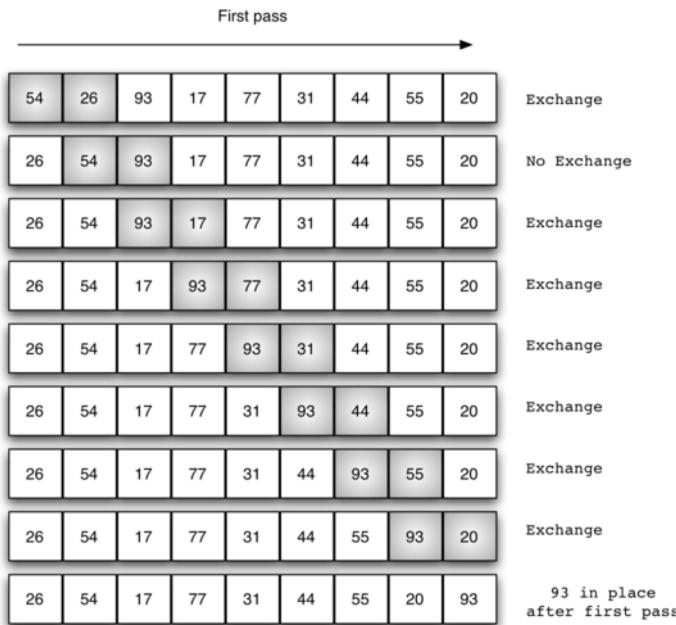


Figure 14.2: One pass for bubble sort

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. 14.2, 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

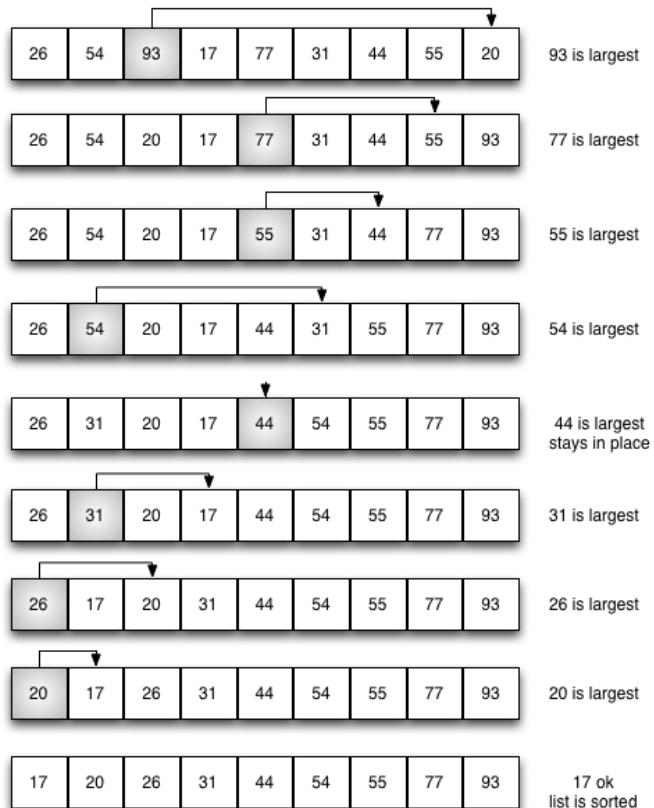


Figure 14.3: The whole process for Selection sort

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 14.3.

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.

14.2 $O(n \log n)$ 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:

$$2^h \geq n!, h \geq \log(n!)h = \Omega(n \log n) \quad (14.1)$$

In 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.

14.2.1 Merge Sort

Merge Sort is the most basic example of divide and conquer strategy, which can improve the complexity from $O(n^2)$ to $O(n \lg n)$. Merge sort there are two main steps: divide and merge (conquer). It is a recursive function, which recursively divide the list into two halves until it reaches out to the base case (only one element) which itself is already a sorted list, then it track back to

the last steps and merge the two lists together and return the merged result to the last function call. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list. Figure 14.4 shows our familiar example list as it is being split by mergeSort. Figure 14.5 shows the simple lists, now sorted, as they are merged back together. The

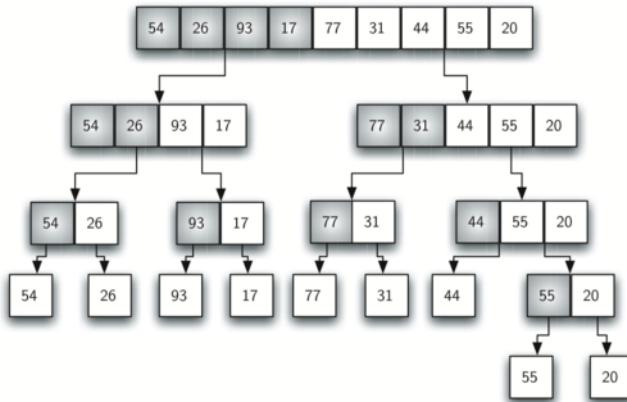


Figure 14.4: Merge Sort: divide process

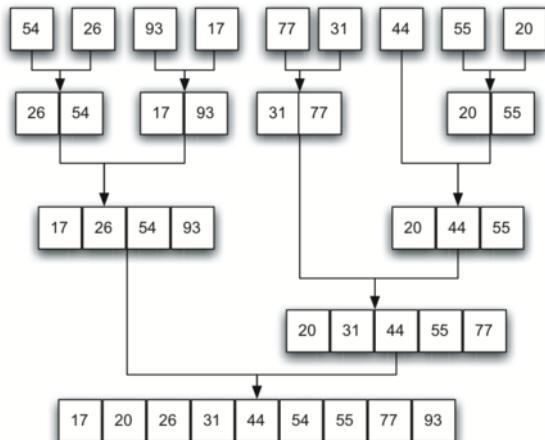


Figure 14.5: Merge Sort: merge process

Python code:

```

1 def mergeSort(nums, reverse = False):
2     if not nums or len(nums) == 1:
3         return nums
4     # divide into two halves
5     mid = len(nums) // 2
6     left = mergeSort(nums[:mid], reverse)
7     right = mergeSort(nums[mid:], reverse)
  
```

```

8
9     # merge
10    tmp = []
11    i, j = 0, 0
12    while i < len(left) and j < len(right):
13        if compare(left[i], right[j], reverse):
14            tmp.append(left[i])
15            i += 1
16        else:
17            tmp.append(right[j])
18            j += 1
19    # deal with the left over elements
20    while i < len(left):
21        tmp.append(left[i])
22        i += 1
23    while j < len(right):
24        tmp.append(right[j])
25        j += 1
26    return tmp

```

As we can see from the code and the merge sort process, each time we divide the list, we made a copy of both halves: `nums[:mid]` and `nums[mid:]`, and when we are merging the list, 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.

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:

$$\begin{aligned}
 T(n) &= 2T(n/2) + O(n) \\
 &= 2 * 2T(n/4) + O(n) + O(n) \\
 &= O(n\log n)
 \end{aligned} \tag{14.2}$$

14.2.2 HeapSort

Implementing heapsort using the `heapq`:

```

1 >>> def heapsort(iterable):
2 ...     h = []
3 ...     for value in iterable:
4 ...         heappush(h, value)
5 ...     return [heappop(h) for i in range(len(h))]
6 ...
7 >>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
8 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

14.2.3 Quick Sort and Quick Select

The quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it

is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the pivot value. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort.

Figure ?? shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The partition process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value. Python code:

```

1 def quickSort(alist):
2     quickSortHelper(alist, 0, len(alist)-1)
3
4 def quickSortHelper(alist, first, last):
5     if first < last:
6
7         splitpoint = partition(alist, first, last)
8
9         quickSortHelper(alist, first, splitpoint -1)
10        quickSortHelper(alist, splitpoint+1, last)
11
12 alist = [54,26,93,17,77,31,44,55,20,100]
13 quickSort(alist)
14 print(alist)
```

Hoare Partition

In Hoare partition, we first pick a pivot, either randomly or left or right end, then we put two indexes at both side of the array, we compared each side's element, if a condition is not satisfied, then stop, we do a switch.

```

1 def HoarePartition(arr, low, high):
2     pivot=arr[high]
3     i=low
4     j=high-1
5     while True:
6         while arr[i] <= pivot and i < j:
7             i += 1
8         while arr[j] >= pivot and i < j:
9             j -= 1
10        if i == j:
11            if arr[i] <= pivot:
12                i += 1
13            arr[i], arr[high] = arr[high], arr[i]
14        return i
```

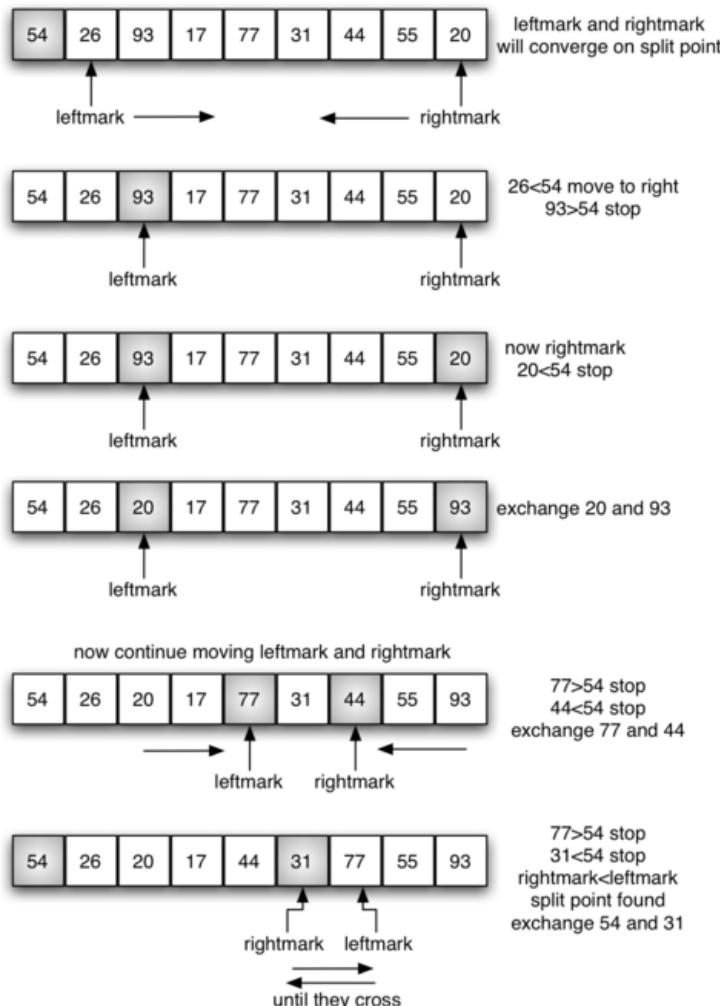


Figure 14.6: Hoare Partition

```

15     else:
16         arr[i], arr[j] = arr[j], arr[i]

```

Lomuto's Partition

For lomuto partition, we first pick the pivot from one side, and the other side is going to be the index for swapping, use index j . Then we iterative index i from j to the pivot side, if the element does not belong to the right side (if pivot starts from left, then the swapping from the last element reversely go to the first element) or left side(if the pivot starts from the rightmost element, then the swapping from the first element to the second last element on the right side), we do a swap. At last, we swap the last j with the pivot position.

```

1 def lumo_partition(alist, first, last):
2     pivotvalue = alist[first]
3     j = last #place for swapping, where j is the smaller element
4         on the right side
5     for i in range(last, first, -1):
6         if alist[i] >= pivotvalue:
7             alist[i], alist[j] = alist[j], alist[i]
8             j-=1 #swap place move
9     #swap the pivot with the last element
10    alist[j], alist[first] = alist[first], alist[j]
11
12    return j

```

There is another way, to put the pivot at the last

```

1 def lumo_partition(alist, first, last):
2     pivotvalue = alist[last]
3     j = first #place for swapping, where j is the smaller
4         element on the right side
5     for i in range(first, last):
6         if alist[i] <= pivotvalue:
7             alist[i], alist[j] = alist[j], alist[i]
8             j+=1 #swap place move
9     #swap the pivot with the last element
10    alist[j], alist[last] = alist[last], alist[j]
11
12    return j

```

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 = lumo_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    return alist[first]

```

```

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
```

14.3 $O(n + k)$ 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

14.4 $O(n)$ 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.

14.4.1 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 .
6 4) Concatenate all sorted buckets .

```

Python code:

14.4.2 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 example:

```

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

```

```

22     for i in range(1,10):
23         count[i] += count[i-1]
24
25     # Build the output array
26     i = n-1
27     while i>=0:
28         index = (arr[i]/exp1)
29         output[ count[(index)%10] - 1 ] = arr[i]
30         count[(index)%10] -= 1
31         i -= 1
32
33     # Copying the output array to arr[],
34     # so that arr now contains sorted numbers
35     i = 0
36     for i in range(0,len(arr)):
37         arr[i] = output[i]
38
39 # Method to do Radix Sort
40 def radixSort(arr):
41
42     # Find the maximum number to know number of digits
43     max1 = max(arr)
44
45     # Do counting sort for every digit. Note that instead
46     # of passing digit number, exp is passed. exp is  $10^i$ 
47     # where i is current digit number
48     exp = 1
49     while max1/exp > 0:
50         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 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.

14.5 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) )

```

14.6 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=...)

```

14.6.1 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.

14.6.2 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))
8
9 student_objects = [Student('john', 'A', 15), Student('jane', 'B',
  12), Student('dave', 'B', 10)]
10 sorted(student_objects, key=lambda x: x.age) # sort by age
11 [(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 `functools` library 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)

```

14.7 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.

¹<https://www.toptal.com/developers/sorting-algorithms>

14.8 LeetCode Problems

Problems

14.1 Insertion Sort List (147). Sort a linked list using insertion sort.

A graphical example of insertion sort. The partial sorted list (black) initially contains only the first element in the list. With each iteration one element (red) is removed from the input data and inserted in-place into the sorted list

Algorithm of Insertion Sort: Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Example 1:

```
Input: 4->2->1->3
Output: 1->2->3->4
```

Example 2:

```
Input: -1->5->3->4->0
Output: -1->0->3->4->5
```

14.2 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.

14.3 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?*

14.4 Largest Number (179, medium).

14.5 **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.*

14.6 148. 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        if pre_node:
21            cur_node.next = next_node
22            pre_node.next = cur_node
23        else:
24            cur_node.next = next_node
25            head = cur_node
26    return sorted_head

```

```
18     cmp_node = cmp_node.next
19
20     if cmp_node == head: #put in the front
21         cur_node.next = head
22         head = cur_node
23     elif cmp_node == None: #put at the back
24         cur_node.next = None #current node is the end,
so set it to None
25         pre_node.next = cur_node
26         #head is not changed
27     else: #in the middle, insert
28         pre_node.next = cur_node
29         cur_node.next = cmp_node
30         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, pre_interal.end
25                 if s <= e_before: #overlap, merge to the
26                     same interval
27                         h[idx].end = max(e, e_before)
28                         bAdd = True
29             if not bAdd:
30                 h.append(i)
```

```

28             break
29     if not bAdd:
30         #no overlap, push to the heap
31         heappush(h, i)
32     return h

```

3. Solution: there could have so many ways to do it, the most easy one is to sort the letters in each string and see if it is the same. Or we can have an array of 26, and save the count of each letter, and check each letter in the other one string.

```

1 def isAnagram(self, s, t):
2     """
3     :type s: str
4     :type t: str
5     :rtype: bool
6     """
7     return ''.join(sorted(list(s))) == ''.join(sorted(
8         list(t)))

```

The second solution is to use a fixed number 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=
LargerNumKey))
8         return '0' if largest_num[0] == '0' else
largest_num
```

Lumo-position.

P
54 26, 93, 17, 77, 31, 44, 55
= J
x ↑

54, 26, 93, 17, 77, 31, 44, 55
↑ j i
↑ j

54, 26, 93, 17, 77, 31, 44, 55
↑ j
i

54, 26, 93, 17, 44, 31, 77, 55
↑ j
i

54, 26, 93, 17, 44, 31, 77, 55
↑ j
i

	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 14.8: The time complexity for common sorting algorithms

15

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.

15.1 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

```

15.2 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 15.1: 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

```

15.3 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.

$$\sum_{i=0}^{N-1} 2^i = 2^{(N-1)} + 2^{(N-2)} + \dots + 2^2 + 2^1 + 2^0 = 2^N - 1 \quad (15.1)$$

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. 15.1.

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:

$$5_{10} = 0000,0101_2, \quad (15.2)$$

$$-5_{10} = 1111,1010_2 + 1_2, \quad (15.3)$$

$$-5_{10} = 1111,1011_2 \quad (15.4)$$

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.

15.4 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. 15.2 and 15.4, 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

```

15.5 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

15.1 **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 15.1.

```
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
```

15.2 **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

```

15.3 421. 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

15.4 190. 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
```

15.5 201. 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
```

15.6 LeetCode Problems

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.

16

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.

16.1 Numbers

16.1.1 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))
```

16.1.2 Ugly Numbers

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. We can write it as $\text{ugly number} = 2^i 3^j 5^k$, $i \geq 0, j \geq 0, k \geq 0$. Examples of ugly numbers: 1, 2, 3, 5, 6, 10, 15, ... The concept of ugly number is quite simple. Now let us use the LeetCode problems as example to derive the algorithms to identify ugly numbers.

Check a Single Number

263. Ugly Number (Easy)

```

1 Ugly numbers are positive numbers whose prime factors only
   include 2, 3, 5. For example, 6, 8 are ugly while 14 is not
   ugly since it includes another prime factor 7.
2
3 Note:
4     1 is typically treated as an ugly number.
5     Input is within the 32-bit signed integer range.

```

Analysis: because the ugly number is only divisible by 2,3,5, so if we keep dividing the number by these factors (num/f), eventually we would get 1, if the remainder ($num\%f$) is 0 (divisible), otherwise we stop the loop to check the number.

```

1 def isUgly(self, num):
2     """
3         :type num: int
4         :rtype: bool
5     """
6     if num == 0:
7         return False
8     factor = [2, 3, 5]
9     for f in factor:
10        while num % f == 0:
11            num /= f
12    return num == 1

```

Generate A Range of Number

264. Ugly Number II (medium)

```

1 Write a program to find the n-th ugly number.
2
3 Ugly numbers are positive numbers whose prime factors only
   include 2, 3, 5. For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12
   is the sequence of the first 10 ugly numbers.
4
5 Note that 1 is typically treated as an ugly number, and n does
   not exceed 1690.

```

Analysis: The first solution is we use the rules $uglynumber = 2^i3^j5^k, i \geq 0, j \geq 0, k \geq 0$, using three for loops to generate at least 1690 ugly numbers that is in the range of 2^{32} , and then sort them, the time complexity is $O(n\log n)$, with $O(n)$ in space. However, if we need to constantly make request, it seems reasonable to save a table, and once the table is generated and saved, each time we would only need constant time to check.

```

1 from math import log, ceil
2 class Solution:
3     ugly = [2**i * 3**j * 5**k for i in range(32) for j in range(
4         (ceil(log(2**32, 3))) for k in range(ceil(log(2**32, 5))))]

```

```

4     ugly.sort()
5     def nthUglyNumber(self, n):
6         """
7             :type n: int
8             :rtype: int
9         """
10    return self.ugly[n-1]

```

The second way is only generate the nth ugly number, with

```

1 class Solution:
2     n = 1690
3     ugly = [1]
4     i2 = i3 = i5 = 0
5     for i in range(n-1):
6         u2, u3, u5 = 2 * ugly[i2], 3 * ugly[i3], 5 * ugly[i5]
7         umin = min(u2, u3, u5)
8         ugly.append(umin)
9         if umin == u2:
10             i2 += 1
11         if umin == u3:
12             i3 += 1
13         if umin == u5:
14             i5 += 1
15
16     def nthUglyNumber(self, n):
17         """
18             :type n: int
19             :rtype: int
20         """
21     return self.ugly[n-1]

```

16.1.3 Combinatorics

1. 611. Valid Triangle Number

16.1 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                 digits[i]
48                 break
49         if stop_index == -1:
50             return -1
51
52         # print(digits)
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

```

16.2 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.

16.2.1 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))

```

16.2.2 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:

$$\text{lcm}(a, b) = \frac{a \times b}{\gcd(a, b)} \quad (16.1)$$

16.3 Arithmetic Operations

Because for the computer, it only understands the binary representation as we learned in Bit Manipulation (Chapter 15, 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 $O(n - m)$.

16.2 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

```

16.4 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.

16.5 Linear Algebra

Gaussian Elimination is one of the several ways to find the solution for a system of linear equations.

16.6 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

```

16.7 Miscellaneous Categories

16.7.1 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.

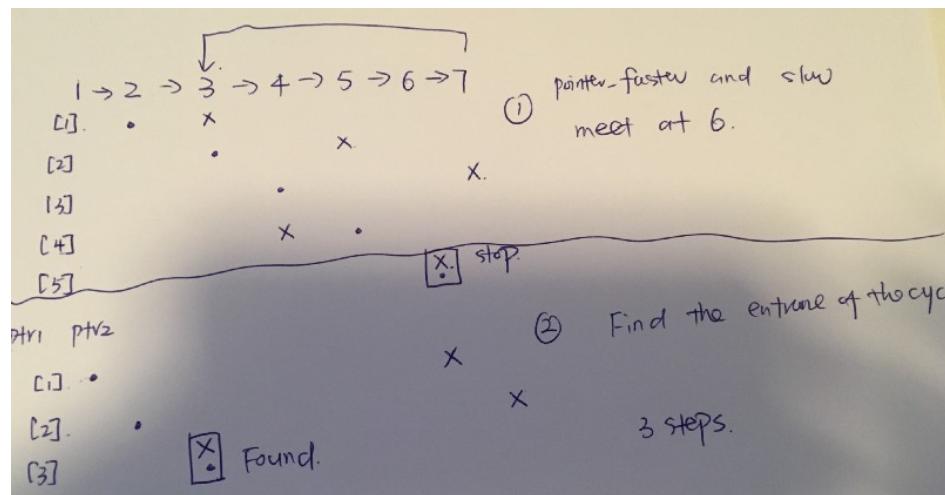


Figure 16.1: Example of floyd's cycle finding

```

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

```

16.8 Exercise

16.8.1 Number

313. Super Ugly Number

¹ Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of size k. For example, [1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32] is the sequence of the first 12 super ugly numbers given primes = [2, 7, 13, 19] of size 4.

²

³ Note:

- ⁴ (1) 1 is a super ugly number for any given primes.
- ⁵ (2) The given numbers in primes are in ascending order.
- ⁶ (3) $0 < k \leq 100$, $0 < n \leq 106$, $0 < \text{primes}[i] < 1000$.
- ⁷ (4) The nth super ugly number is guaranteed to fit in a 32-bit signed integer.

```

1 def nthSuperUglyNumber(self, n, primes):
2     """
3         :type n: int
4         :type primes: List[int]
5         :rtype: int
6     """
7     nums=[1]
8     idexs=[0]*len(primes) #first is the current index
9     for i in range(n-1):
10         min_v = maxsize
11         min_j = []
12         for j, index in enumerate(idexs):
13             v = nums[index]*primes[j]
14             if v<min_v:
15                 min_v = v
16                 min_j=[j]
17             elif v==min_v:
18                 min_j.append(j) #we can get multiple j if
there is a tie

```

```
19         nums.append(min_v)
20         for j in min_j:
21             indexs[j]+=1
22     return nums[-1]
```

Part VII

Appendix

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.

17.1 Python Overview

In this section, we provide a well-organized overview of how Python works as an **object-oriented programming language** in Subsection 17.1.1, what is the components of Python: built-in Data types, built-in modules, third party packages/libraries, frameworks (Subsection 17.1.2). And in this book, we selectively introduce the most useful ones that considered for the purpose of learning algorithms and passing coding interviews.

17.1.1 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 '''

```

17.1.2 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 17.9 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.

17.2 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 15), and two special operators like the identity operator or the membership operator.

17.2.1 Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Table 17.1: 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

17.2.2 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`.

17.2.3 Comparison Operators

Comparison operators are used to compare values. It either returns True or False according to the condition.

Table 17.2: 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>

17.2.4 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 17.3: 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

17.2.5 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 17.4: 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 17.5: 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

17.3 Function

17.3.1 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.

17.3.2 Lambda Function

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
```

17.3.3 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

```

17.4 Class

17.4.1 Special Methods

From [1]. <http://www.informit.com/articles/article.aspx?p=453682&seqNum=6> All the built-in data types implement a collection of special object methods. The names of special methods are always preceded and followed by double underscores (`__`). These methods are automatically triggered by the interpreter as a program executes. For example, the operation $x + y$ is mapped to an internal method, `x.__add__(y)`, and an indexing operation, `x[k]`, is mapped to `x.__getitem__(k)`. The behavior of each data type depends entirely on the set of special methods that it implements.

User-defined classes can define new objects that behave like the built-in types simply by supplying an appropriate subset of the special methods described in this section. In addition, built-in types such as lists and dictionaries can be specialized (via inheritance) by redefining some of the special methods. In this book, we only list the essential ones so that it speeds up our interview preparation.

Object Creation, Destruction, and Representation We first list these special methods in Table 17.6. A good and useful way to imple-

Table 17.6: 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 17.7 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 17.7: 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>

17.4.2 Class Syntax

17.4.3 Inheritance

17.4.4 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()`.

17.5 Shallow Copy and the deep copy

For list and string data structures, we constantly met the case that we need to copy. However, in programming language like C++, Python, we need to know the difference between shallow copy and deep copy. Here we only introduce the Python version.

Given the following two snippets of Python code:

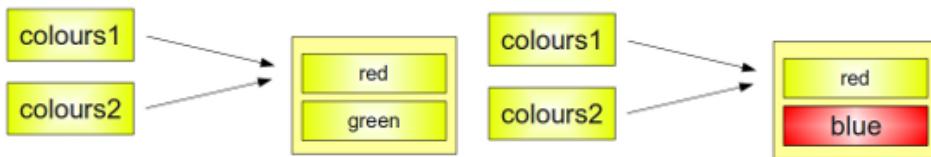
```

1 colours1 = [ "red" , "green" ]
2 colours2 = colours1
3 colours2 = [ "rouge" , "vert" ]
4 print(colours1)
5 >>> [ 'red' , 'green' ]
```

```

1 colours1 = [ "red" , "green" ]
2 colours2 = colours1
3 colours2[1] = "blue"
4 print(colours1)
5 [ 'red' , 'blue' ]
```

From the above outputs, we can see that the colors1 list is the same but in the second case, it is changed although we are assigning value to colors2. The result can be either wanted or not wanted. In python, to assign one list to other directly is similar to a pointer in C++, which both point to the same physical address. In the first case, colors2 is reassigned a new list, which has a new address, so now colors2 points to the address of this new list instead, which leaves the values of colors2 untouched at all. We can visualize this process as follows: However, we often need to do copy and



(a) The copy process for code 1

(b) The copy process for code 2

Figure 17.1: Copy process

leave the original list or string unchanged. Because there are a variety of list, from one dimensional, two-dimensional to multi-dimensional.

17.5.1 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:

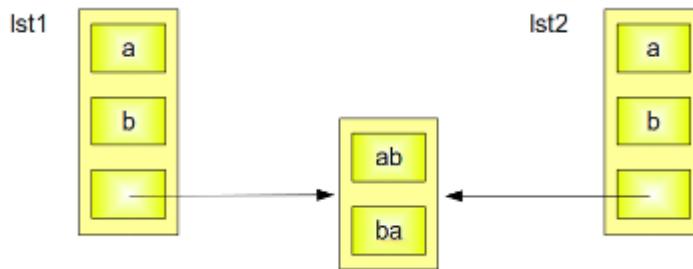


Figure 17.2: Caption

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.

17.5.2 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

```

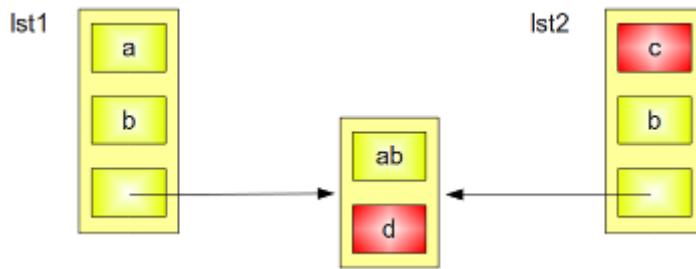


Figure 17.3: Caption

```

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']]

```

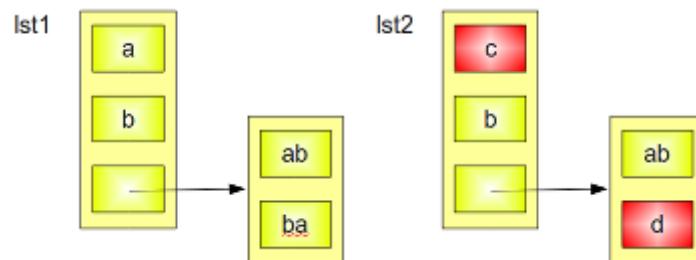


Figure 17.4: Caption

This section is cited from Need to modify.

17.6 Global Vs nonlocal

17.7 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)

```

17.8 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.

17.9 Supplemental Python Tools

17.9.1 Re

17.9.2 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

```

17.9.3 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 17.8. 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 17.8: 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].