ELEMENTS OF PROGRAMMING INTERVIEWS IN python™

ADNAN AZIZ
TSUNG-HSIEN LEE
AMIT PRAKASH

Elements of Programming Interviews in Python

The Insiders' Guide

Adnan Aziz
Tsung-Hsien Lee
Amit Prakash

ElementsOfProgrammingInterviews.com

Adnan Aziz is a Research Scientist at Facebook, where his team develops the technology that powers everything from check-ins to Facebook Pages. Formerly, he was a professor at the Department of Electrical and Computer Engineering at The University of Texas at Austin, where he conducted research and taught classes in applied algorithms. He received his Ph.D. from The University of California at Berkeley; his undergraduate degree is from Indian Institutes of Technology Kanpur. He has worked at Google, Qualcomm, IBM, and several software startups. When not designing algorithms, he plays with his children, Laila, Imran, and Omar.

Tsung-Hsien Lee is a Senior Software Engineer at Uber working on self-driving cars. Previously, he worked as a Software Engineer at Google and as Software Engineer Intern at Facebook. He received both his M.S. and undergraduate degrees from National Tsing Hua University. He has a passion for designing and implementing algorithms. He likes to apply algorithms to every aspect of his life. He takes special pride in helping to organize Google Code Jam 2014 and 2015.

Amit Prakash is a co-founder and CTO of ThoughtSpot, a Silicon Valley startup. Previously, he was a Member of the Technical Staff at Google, where he worked primarily on machine learning problems that arise in the context of online advertising. Before that he worked at Microsoft in the web search team. He received his Ph.D. from The University of Texas at Austin; his undergraduate degree is from Indian Institutes of Technology Kanpur. When he is not improving business intelligence, he indulges in his passion for puzzles, movies, travel, and adventures with Nidhi and Aanya.

Elements of Programming Interviews in Python: The Insiders' Guide
by Adnan Aziz, Tsung-Hsien Lee, and Amit Prakash

We typeset this book using LaTeX and the Memoir class. We used TikZ to draw figures. Allan Ytac created the cover, based on a design brief we provided.

The companion website for the book includes contact information and a list of known errors for each version of the book. If you come across an error or an improvement, please let us know.

Website: http://elementsofprogramminginterviews.com

To my father, Ishrat Aziz,
for giving me my lifelong love of learning

Adnan Aziz

To my parents, Hsien-Kuo Lee and Tseng-Hsia Li,
for the everlasting support and love they give me

Tsung-Hsien Lee

To my parents, Manju Shree and Arun Prakash,
the most loving parents I can imagine

Amit Prakash

Table of Contents

Introduction 1

I The Interview 5

1 Getting Ready 6

2 Strategies For A Great Interview 13

3 Conducting An Interview 19

II Data Structures and Algorithms 22

4 Primitive Types 23
- 4.1 Computing the parity of a word . . . 24
- 4.2 Swap bits . . . 27
- 4.3 Reverse bits . . . 28
- 4.4 Find a closest integer with the same weight . . . 28
- 4.5 Compute $x \times y$ without arithmetical operators . . . 29
- 4.6 Compute x/y . . . 31
- 4.7 Compute x^y . . . 32
- 4.8 Reverse digits . . . 32
- 4.9 Check if a decimal integer is a palindrome . . . 33
- 4.10 Generate uniform random numbers . . . 34
- 4.11 Rectangle intersection . . . 35

5 Arrays 37
- 5.1 The Dutch national flag problem . . . 39
- 5.2 Increment an arbitrary-precision integer . . . 43
- 5.3 Multiply two arbitrary-precision integers . . . 43
- 5.4 Advancing through an array . . . 44
- 5.5 Delete duplicates from a sorted array . . . 45
- 5.6 Buy and sell a stock once . . . 46
- 5.7 Buy and sell a stock twice . . . 47

5.8 Computing an alternation . . . 48
5.9 Enumerate all primes to n . . . 49
5.10 Permute the elements of an array . . . 50
5.11 Compute the next permutation . . . 52
5.12 Sample offline data . . . 54
5.13 Sample online data . . . 55
5.14 Compute a random permutation . . . 56
5.15 Compute a random subset . . . 57
5.16 Generate nonuniform random numbers . . . 58
5.17 The Sudoku checker problem . . . 60
5.18 Compute the spiral ordering of a 2D array . . . 61
5.19 Rotate a 2D array . . . 64
5.20 Compute rows in Pascal's Triangle . . . 65

6 Strings **67**
6.1 Interconvert strings and integers . . . 68
6.2 Base conversion . . . 69
6.3 Compute the spreadsheet column encoding . . . 70
6.4 Replace and remove . . . 71
6.5 Test palindromicity . . . 72
6.6 Reverse all the words in a sentence . . . 73
6.7 Compute all mnemonics for a phone number . . . 74
6.8 The look-and-say problem . . . 75
6.9 Convert from Roman to decimal . . . 76
6.10 Compute all valid IP addresses . . . 77
6.11 Write a string sinusoidally . . . 78
6.12 Implement run-length encoding . . . 79
6.13 Find the first occurrence of a substring . . . 79

7 Linked Lists **82**
7.1 Merge two sorted lists . . . 84
7.2 Reverse a single sublist . . . 85
7.3 Test for cyclicity . . . 86
7.4 Test for overlapping lists—lists are cycle-free . . . 87
7.5 Test for overlapping lists—lists may have cycles . . . 88
7.6 Delete a node from a singly linked list . . . 90
7.7 Remove the kth last element from a list . . . 90
7.8 Remove duplicates from a sorted list . . . 91
7.9 Implement cyclic right shift for singly linked lists . . . 92
7.10 Implement even-odd merge . . . 93
7.11 Test whether a singly linked list is palindromic . . . 94
7.12 Implement list pivoting . . . 95
7.13 Add list-based integers . . . 96

8 Stacks and Queues **97**
8.1 Implement a stack with max API . . . 98
8.2 Evaluate RPN expressions . . . 101

8.3 Test a string over "{,},(,),[,]" for well-formedness . . . 102
8.4 Normalize pathnames . . . 102
8.5 Compute buildings with a sunset view . . . 103
8.6 Compute binary tree nodes in order of increasing depth . . . 106
8.7 Implement a circular queue . . . 107
8.8 Implement a queue using stacks . . . 108
8.9 Implement a queue with max API . . . 109

9 Binary Trees **112**
9.1 Test if a binary tree is height-balanced . . . 114
9.2 Test if a binary tree is symmetric . . . 116
9.3 Compute the lowest common ancestor in a binary tree . . . 117
9.4 Compute the LCA when nodes have parent pointers . . . 118
9.5 Sum the root-to-leaf paths in a binary tree . . . 119
9.6 Find a root to leaf path with specified sum . . . 120
9.7 Implement an inorder traversal without recursion . . . 121
9.8 Implement a preorder traversal without recursion . . . 121
9.9 Compute the kth node in an inorder traversal . . . 122
9.10 Compute the successor . . . 123
9.11 Implement an inorder traversal with $O(1)$ space . . . 124
9.12 Reconstruct a binary tree from traversal data . . . 125
9.13 Reconstruct a binary tree from a preorder traversal with markers . . . 127
9.14 Form a linked list from the leaves of a binary tree . . . 128
9.15 Compute the exterior of a binary tree . . . 128
9.16 Compute the right sibling tree . . . 129

10 Heaps **132**
10.1 Merge sorted files . . . 134
10.2 Sort an increasing-decreasing array . . . 135
10.3 Sort an almost-sorted array . . . 136
10.4 Compute the k closest stars . . . 137
10.5 Compute the median of online data . . . 139
10.6 Compute the k largest elements in a max-heap . . . 140

11 Searching **142**
11.1 Search a sorted array for first occurrence of k . . . 145
11.2 Search a sorted array for entry equal to its index . . . 146
11.3 Search a cyclically sorted array . . . 147
11.4 Compute the integer square root . . . 148
11.5 Compute the real square root . . . 149
11.6 Search in a 2D sorted array . . . 150
11.7 Find the min and max simultaneously . . . 152
11.8 Find the kth largest element . . . 153
11.9 Find the missing IP address . . . 155
11.10 Find the duplicate and missing elements . . . 157

12 Hash Tables **159**

12.1 Test for palindromic permutations . . . 163
12.2 Is an anonymous letter constructible? . . . 164
12.3 Implement an ISBN cache . . . 165
12.4 Compute the LCA, optimizing for close ancestors . . . 166
12.5 Find the nearest repeated entries in an array . . . 167
12.6 Find the smallest subarray covering all values . . . 168
12.7 Find smallest subarray sequentially covering all values . . . 171
12.8 Find the longest subarray with distinct entries . . . 173
12.9 Find the length of a longest contained interval . . . 174
12.10 Compute all string decompositions . . . 175
12.11 Test the Collatz conjecture . . . 176
12.12 Implement a hash function for chess . . . 177

13 Sorting **180**
13.1 Compute the intersection of two sorted arrays . . . 182
13.2 Merge two sorted arrays . . . 183
13.3 Remove first-name duplicates . . . 184
13.4 Smallest nonconstructible value . . . 185
13.5 Render a calendar . . . 186
13.6 Merging intervals . . . 188
13.7 Compute the union of intervals . . . 189
13.8 Partitioning and sorting an array with many repeated entries . . . 191
13.9 Team photo day—1 . . . 193
13.10 Implement a fast sorting algorithm for lists . . . 194
13.11 Compute a salary threshold . . . 195

14 Binary Search Trees **197**
14.1 Test if a binary tree satisfies the BST property . . . 199
14.2 Find the first key greater than a given value in a BST . . . 201
14.3 Find the k largest elements in a BST . . . 202
14.4 Compute the LCA in a BST . . . 203
14.5 Reconstruct a BST from traversal data . . . 204
14.6 Find the closest entries in three sorted arrays . . . 206
14.7 Enumerate numbers of the form $a + b\sqrt{2}$ . . . 207
14.8 Build a minimum height BST from a sorted array . . . 210
14.9 Test if three BST nodes are totally ordered . . . 211
14.10 The range lookup problem . . . 212
14.11 Add credits . . . 215

15 Recursion **217**
15.1 The Towers of Hanoi problem . . . 219
15.2 Generate all nonattacking placements of n-Queens . . . 221
15.3 Generate permutations . . . 222
15.4 Generate the power set . . . 224
15.5 Generate all subsets of size k . . . 226
15.6 Generate strings of matched parens . . . 227
15.7 Generate palindromic decompositions . . . 228

15.8 Generate binary trees . . . 229
15.9 Implement a Sudoku solver . . . 230
15.10 Compute a Gray code . . . 232

16 Dynamic Programming **234**
16.1 Count the number of score combinations . . . 236
16.2 Compute the Levenshtein distance . . . 239
16.3 Count the number of ways to traverse a 2D array . . . 242
16.4 Compute the binomial coefficients . . . 244
16.5 Search for a sequence in a 2D array . . . 245
16.6 The knapsack problem . . . 246
16.7 The bedbathandbeyond.com problem . . . 249
16.8 Find the minimum weight path in a triangle . . . 251
16.9 Pick up coins for maximum gain . . . 252
16.10 Count the number of moves to climb stairs . . . 253
16.11 The pretty printing problem . . . 254
16.12 Find the longest nondecreasing subsequence . . . 257

17 Greedy Algorithms and Invariants **259**
17.1 Compute an optimum assignment of tasks . . . 260
17.2 Schedule to minimize waiting time . . . 261
17.3 The interval covering problem . . . 262
17.4 The 3-sum problem . . . 264
17.5 Find the majority element . . . 266
17.6 The gasup problem . . . 267
17.7 Compute the maximum water trapped by a pair of vertical lines . . . 269
17.8 Compute the largest rectangle under the skyline . . . 270

18 Graphs **273**
18.1 Search a maze . . . 276
18.2 Paint a Boolean matrix . . . 278
18.3 Compute enclosed regions . . . 280
18.4 Deadlock detection . . . 281
18.5 Clone a graph . . . 282
18.6 Making wired connections . . . 283
18.7 Transform one string to another . . . 284
18.8 Team photo day—2 . . . 286

19 Parallel Computing **289**
19.1 Implement caching for a multithreaded dictionary . . . 291
19.2 Analyze two unsynchronized interleaved threads . . . 292
19.3 Implement synchronization for two interleaving threads . . . 293
19.4 Implement a thread pool . . . 294
19.5 Deadlock . . . 295
19.6 The readers-writers problem . . . 296
19.7 The readers-writers problem with write preference . . . 297
19.8 Implement a Timer class . . . 298

19.9 Test the Collatz conjecture in parallel 298

III Domain Specific Problems **300**

20 Design Problems **301**
20.1 Design a spell checker 302
20.2 Design a solution to the stemming problem 303
20.3 Plagiarism detector 304
20.4 Pair users by attributes 305
20.5 Design a system for detecting copyright infringement 306
20.6 Design TEX 307
20.7 Design a search engine 307
20.8 Implement PageRank 308
20.9 Design TeraSort and PetaSort 310
20.10 Implement distributed throttling 310
20.11 Design a scalable priority system 311
20.12 Create photomosaics 312
20.13 Implement Mileage Run 312
20.14 Implement Connexus 314
20.15 Design an online advertising system 315
20.16 Design a recommendation system 316
20.17 Design an optimized way of distributing large files 316
20.18 Design the World Wide Web 317
20.19 Estimate the hardware cost of a photo sharing app 318

21 Language Questions **319**
21.1 Garbage Collection 319
21.2 Closure 319
21.3 Shallow and deep copy 320
21.4 Iterators and Generators 321
21.5 `@decorator` 321
21.6 List vs tuple 323
21.7 `*args` and `*kwargs` 324
21.8 Python code 325
21.9 Exception Handling 326
21.10 Scoping 328
21.11 Function arguments 330

22 Object-Oriented Design **333**
22.1 Template Method vs. Strategy 333
22.2 Observer pattern 334
22.3 Push vs. pull observer pattern 334
22.4 Singletons and Flyweights 335
22.5 Adapters 336
22.6 Creational Patterns 337
22.7 Libraries and design patterns 338

23 Common Tools 339
23.1 Merging in a version control system . . . 339
23.2 Hooks . . . 341
23.3 Is scripting more efficient? . . . 342
23.4 Polymorphism with a scripting language . . . 343
23.5 Dependency analysis . . . 343
23.6 ANT vs. Maven . . . 344
23.7 SQL vs. NoSQL . . . 345
23.8 Normalization . . . 345
23.9 SQL design . . . 346
23.10 IP, TCP, and HTTP . . . 346
23.11 HTTPS . . . 347
23.12 DNS . . . 348

IV The Honors Class 349

24 Honors Class 350
24.1 Compute the greatest common divisor . . . 351
24.2 Find the first missing positive entry . . . 352
24.3 Buy and sell a stock k times . . . 353
24.4 Compute the maximum product of all entries but one . . . 354
24.5 Compute the longest contiguous increasing subarray . . . 356
24.6 Rotate an array . . . 357
24.7 Identify positions attacked by rooks . . . 359
24.8 Justify text . . . 360
24.9 Implement list zipping . . . 361
24.10 Copy a postings list . . . 362
24.11 Compute the longest substring with matching parens . . . 364
24.12 Compute the maximum of a sliding window . . . 365
24.13 Implement a postorder traversal without recursion . . . 366
24.14 Compute fair bonuses . . . 368
24.15 Search a sorted array of unknown length . . . 370
24.16 Search in two sorted arrays . . . 372
24.17 Find the kth largest element—large n, small k . . . 373
24.18 Find an element that appears only once . . . 374
24.19 Find the line through the most points . . . 375
24.20 Convert a sorted doubly linked list into a BST . . . 376
24.21 Convert a BST to a sorted doubly linked list . . . 378
24.22 Merge two BSTs . . . 379
24.23 Implement regular expression matching . . . 380
24.24 Synthesize an expression . . . 383
24.25 Count inversions . . . 385
24.26 Draw the skyline . . . 386
24.27 Measure with defective jugs . . . 388
24.28 Compute the maximum subarray sum in a circular array . . . 390

24.29 Determine the critical height  391
24.30 Find the maximum 2D subarray  393
24.31 Implement Huffman coding  395
24.32 Trapping water 398
24.33 The heavy hitter problem 399
24.34 Find the longest subarray whose sum $\leq k$  400
24.35 Road network 402
24.36 Test if arbitrage is possible  403

V Notation and Index **406**

Notation **407**

Index of Terms **409**

Introduction

It's not that I'm so smart, it's just that I stay with problems longer.

— A. Einstein

Elements of Programming Interviews (EPI) aims to help engineers interviewing for software development positions. The primary focus of EPI is data structures, algorithms, system design, and problem solving. The material is largely presented through questions.

An interview problem

Let's begin with Figure 1 below. It depicts movements in the share price of a company over 40 days. Specifically, for each day, the chart shows the daily high and low, and the price at the opening bell (denoted by the white square). Suppose you were asked in an interview to design an algorithm that determines the maximum profit that could have been made by buying and then selling a single share over a given day range, subject to the constraint that the buy and the sell have to take place at the start of the day. (This algorithm may be needed to backtest a trading strategy.)

You may want to stop reading now, and attempt this problem on your own.

First clarify the problem. For example, you should ask for the input format. Let's say the input consists of three arrays L, H, and S, of nonnegative floating point numbers, representing the low, high, and starting prices for each day. The constraint that the purchase and sale have to take place at the start of the day means that it suffices to consider S. You may be tempted to simply return the difference of the minimum and maximum elements in S. If you try a few test cases, you will see that the minimum can occur after the maximum, which violates the requirement in the problem statement—you have to buy before you can sell.

At this point, a brute-force algorithm would be appropriate. For each pair of indices i and $j > i$, if $S[j] - S[i]$ is greater than the largest difference seen so far, update the largest difference to $S[j] - S[i]$. You should be able to code this algorithm using a pair of nested for-loops and test

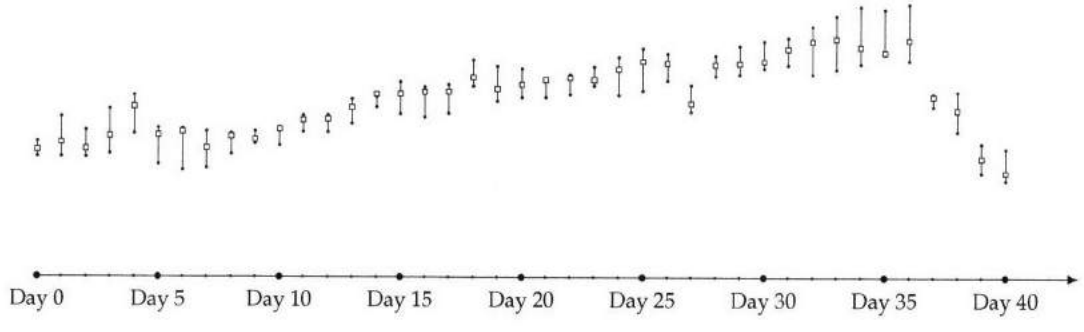


Figure 1: Share price as a function of time.

it in a matter of a few minutes. You should also derive its time complexity as a function of the length n of the input array. The outer loop is invoked $n-1$ times, and the ith iteration processes $n-1-i$ elements. Processing an element entails computing a difference, performing a compare, and possibly updating a variable, all of which take constant time. Hence, the run time is proportional to $\sum_{i=0}^{n-2}(n-1-i) = \frac{(n-1)(n)}{2}$, i.e., the time complexity of the brute-force algorithm is $O(n^2)$. You should also consider the space complexity, i.e., how much memory your algorithm uses. The array itself takes memory proportional to n, and the additional memory used by the brute-force algorithm is a constant independent of n—a couple of iterators and one floating point variable.

Once you have a working algorithm, try to improve upon it. Specifically, an $O(n^2)$ algorithm is usually not acceptable when faced with large arrays. You may have heard of an algorithm design pattern called divide-and-conquer. It yields the following algorithm for this problem. Split S into two subarrays, $S[0, \lfloor \frac{n}{2} \rfloor]$ and $S[\lfloor \frac{n}{2} \rfloor + 1, n-1]$; compute the best result for the first and second subarrays; and combine these results. In the combine step we take the better of the results for the two subarrays. However, we also need to consider the case where the optimum buy and sell take place in separate subarrays. When this is the case, the buy must be in the first subarray, and the sell in the second subarray, since the buy must happen before the sell. If the optimum buy and sell are in different subarrays, the optimum buy price is the minimum price in the first subarray, and the optimum sell price is in the maximum price in the second subarray. We can compute these prices in $O(n)$ time with a single pass over each subarray. Therefore, the time complexity $T(n)$ for the divide-and-conquer algorithm satisfies the recurrence relation $T(n) = 2T(\frac{n}{2}) + O(n)$, which solves to $O(n \log n)$.

The divide-and-conquer algorithm is elegant and fast. Its implementation entails some corner cases, e.g., an empty subarray, subarrays of length one, and an array in which the price decreases monotonically, but it can still be written and tested by a good developer in 20–30 minutes.

Looking carefully at the combine step of the divide-and-conquer algorithm, you may have a flash of insight. Specifically, you may notice that the maximum profit that can be made by selling on a specific day is determined by the minimum of the stock prices over the previous days. Since the maximum profit corresponds to selling on *some* day, the following algorithm correctly computes the maximum profit. Iterate through S, keeping track of the minimum element m seen thus far. If the difference of the current element and m is greater than the maximum profit recorded so far, update the maximum profit. This algorithm performs a constant amount of work per array element, leading to an $O(n)$ time complexity. It uses two float-valued variables (the minimum element and the maximum profit recorded so far) and an iterator, i.e., $O(1)$ additional space. It is considerably simpler to implement than the divide-and-conquer algorithm—a few minutes should suffice to write and test it. Working code is presented in Solution 5.6 on Page 47.

If in a 45–60 minutes interview, you can develop the algorithm described above, implement and test it, and analyze its complexity, you would have had a very successful interview. In particular, you would have demonstrated to your interviewer that you possess several key skills:

- The ability to rigorously formulate real-world problems.
- The skills to solve problems and design algorithms.
- The tools to go from an algorithm to a tested program.
- The analytical techniques required to determine the computational complexity of your solution.

Book organization

Interviewing successfully is about more than being able to intelligently select data structures and design algorithms quickly. For example, you also need to know how to identify suitable compa-

nies, pitch yourself, ask for help when you are stuck on an interview problem, and convey your enthusiasm. These aspects of interviewing are the subject of Chapters 1–3, and are summarized in Table 1.1 on Page 7.

Chapter 1 is specifically concerned with preparation. Chapter 2 discusses how you should conduct yourself at the interview itself. Chapter 3 describes interviewing from the interviewer's perspective. The latter is important for candidates too, because of the insights it offers into the decision making process.

Since not everyone will have the time to work through EPI in its entirety, we have prepared a study guide (Table 1.2 on Page 8) to problems you should solve, based on the amount of time you have available.

The problem chapters are organized as follows. Chapters 4–14 are concerned with basic data structures, such as arrays and binary search trees, and basic algorithms, such as binary search and quicksort. In our experience, this is the material that most interview questions are based on. Chapters 15–18 cover advanced algorithm design principles, such as dynamic programming and heuristics, as well as graphs. Chapter 19 focuses on parallel programming.

Each chapter begins with an introduction followed by problems. The introduction itself consists of a brief review of basic concepts and terminology, followed by a boot camp. Each boot camp is (1.) a straightforward, illustrative example that illustrates the essence of the chapter without being too challenging; and (2.) top tips for the subject matter, presented in tabular format. For chapters where the programming language includes features that are relevant, we present these features in list form. This list is ordered with basic usage coming first, followed by subtler aspects. Basic usage is demonstrated using methods calls with concrete arguments, e.g., `D = collections.OrderedDict((1,2), (3,4))`. Subtler aspects of the library, such as ways to reduce code length, underappreciated features, and potential pitfalls, appear later in the list. Broadly speaking, the problems are ordered by subtopic, with more commonly asked problems appearing first. Chapter 24 consists of a collection of more challenging problems.

Domain-specific knowledge is covered in Chapters 20,21,22, and 23, which are concerned with system design, programming language concepts, object-oriented programming, and commonly used tools. Keep in mind that some companies do not ask these questions—you should investigate the topics asked by companies you are interviewing at before investing too much time in them. These problems are more likely to be asked of architects, senior developers and specialists.

The notation, specifically the symbols we use for describing algorithms, e.g., $\sum_{i=0}^{n-1} i^2, [a, b), \langle 2, 3, 5, 7\rangle, A[i, j], \lceil x \rceil, (1011)_2, n!, \{x \mid x^2 > 2\}$, etc., is summarized starting on Page 407. It should be familiar to anyone with a technical undergraduate degree, but we still request you to review it carefully before getting into the book, and whenever you have doubts about the meaning of a symbol. Terms, e.g., BFS and dequeue, are indexed starting on Page 409.

The EPI editorial style

Solutions are based on basic concepts, such as arrays, hash tables, and binary search, used in clever ways. Some solutions use relatively advanced machinery, e.g., Dijkstra's shortest path algorithm. You will encounter such problems in an interview only if you have a graduate degree or claim specialized knowledge.

Most solutions include code snippets. Please read Section 1 on Page 11 to familiarize yourself with the Python constructs and practices used in this book. Source code, which includes randomized and directed test cases, can be found at the book website. Domain specific problems are conceptual and not meant to be coded; a few algorithm design problems are also in this spirit.

One of our key design goals for EPI was to make learning easier by establishing a uniform way in which to describe problems and solutions. We refer to our exposition style as the EPI Editorial Style.

Problems are specified as follows:

(1.) We establish **context**, e.g., a real-world scenario, an example, etc.

(2.) We **state the problem** to be solved. Unlike a textbook, but as is true for an interview, we do not give formal specifications, e.g., we do not specify the detailed input format or say what to do on illegal inputs. As a general rule, avoid writing code that parses input. See Page 14 for an elaboration.

(3.) We give a **short hint**—you should read this only if you get stuck. (The hint is similar to what an interviewer will give you if you do not make progress.)

Solutions are developed as follows:

(1.) We begin a **simple brute-force solution.**

(2.) We then **analyze** the brute-force approach and try to get **intuition** for why it is **inefficient** and where we can **improve upon it**, possibly by looking at concrete examples, related algorithms, etc.

(3.) Based on these insights, we develop a **more efficient** algorithm, and describe it in prose.

(4.) We **apply** the program to a concrete input.

(5.) We give **code** for the key steps.

(6.) We analyze time and space **complexity**.

(7.) We outline **variants**—problems whose formulation or solution is similar to the solved problem. Use variants for practice, and to test your understanding of the solution.

Note that exceptions exists to this style—for example a brute-force solution may not be meaningful, e.g., if it entails enumerating all double-precision floating point numbers in some range. For the chapters at the end of the book, which correspond to more advanced topics, such as Dynamic Programming, and Graph Algorithms, we use more parsimonious presentations, e.g., we forgo examples of applying the derived algorithm to a concrete example.

Level and prerequisites

We expect readers to be familiar with data structures and algorithms taught at the undergraduate level. The chapters on concurrency and system design require knowledge of locks, distributed systems, operating systems (OS), and insight into commonly used applications. Some of the material in the later chapters, specifically dynamic programming, graphs, and greedy algorithms, is more advanced and geared towards candidates with graduate degrees or specialized knowledge.

The review at the start of each chapter is not meant to be comprehensive and if you are not familiar with the material, you should first study it in an algorithms textbook. There are dozens of such texts and our preference is to master one or two good books rather than superficially sample many. *Algorithms* by Dasgupta, *et al.* is succinct and beautifully written; *Introduction to Algorithms* by Cormen, *et al.* is an amazing reference.

Reader engagement

Many of the best ideas in EPI came from readers like you. The study guide, ninja notation, and hints, are a few examples of many improvements that were brought about by our readers. The companion website, elementsofprogramminginterviews.com, includes a Stack Overflow-style discussion forum, and links to our social media presence. It also has links blog postings, code, and bug reports. You can always communicate with us directly—our contact information is on the website.

Part I

The Interview

CHAPTER

1 Getting Ready

Before everything else, getting ready is the secret of success.

— H. FORD

The most important part of interview preparation is knowing the material and practicing problem solving. However, the nontechnical aspects of interviewing are also very important, and often overlooked. Chapters 1–3 are concerned with the nontechnical aspects of interviewing, ranging from résumé preparation to how hiring decisions are made. These aspects of interviewing are summarized in Table 1.1 on the facing page

Study guide

Ideally, you would prepare for an interview by solving all the problems in EPI. This is doable over 12 months if you solve a problem a day, where solving entails writing a program and getting it to work on some test cases.

Since different candidates have different time constraints, we have outlined several study scenarios, and recommended a subset of problems for each scenario. This information is summarized in Table 1.2 on Page 8. The preparation scenarios we consider are Hackathon (a weekend entirely devoted to preparation), finals cram (one week, 3–4 hours per day), term project (four weeks, 1.5–2.5 hours per day), and algorithms class (3–4 months, 1 hour per day).

A large majority of the interview questions at Google, Amazon, Microsoft, and similar companies are drawn from the topics in Chapters 4–14. Exercise common sense when using Table 1.2, e.g., if you are interviewing for a position with a financial firm, do more problems related to probability.

Although an interviewer may occasionally ask a question directly from EPI, you should not base your preparation on memorizing solutions. Rote learning will likely lead to your giving a perfect solution to the wrong problem.

Chapter 24 contains a diverse collection of challenging questions. Use them to hone your problem solving skills, but go to them only after you have made major inroads into the earlier chapters. If you have a graduate degree, or claim specialized knowledge, you should definitely solve some problems from Chapter 24.

The interview lifecycle

Generally speaking, interviewing takes place in the following steps:

(1.) Identify companies that you are interested in, and, ideally, find people you know at these companies.

(2.) Prepare your résumé using the guidelines on Page 8, and submit it via a personal contact (preferred), or through an online submission process or a campus career fair.

Table 1.1: A summary of nontechnical aspects of interviewing

The Interview Lifecycle, on the preceding page

- Identify companies, contacts
- Résumé preparation
  - ◇ Basic principles
  - ◇ Website with links to projects
  - ◇ LinkedIn profile & recommendations
- Résumé submission
- Mock interview practice
- Phone/campus screening
- On-site interview
- Negotiating an offer

At the Interview, on Page 13

- Don't solve the wrong problem
- Get specs & requirements
- Construct sample input/output
- Work on concrete examples first
- Spell out the brute-force solution
- Think out loud
- Apply patterns
- Assume valid inputs
- Test for corner-cases
- Use proper syntax
- Manage the whiteboard
- Be aware of memory management
- Get function signatures right

General Advice, on Page 16

- Know the company & interviewers
- Communicate clearly
- Be passionate
- Be honest
- Stay positive
- Don't apologize
- Leave perks and money out
- Be well-groomed
- Mind your body language
- Be ready for a stress interview
- Learn from bad outcomes
- Negotiate the best offer

Conducting an Interview, on Page 19

- Don't be indecisive
- Create a brand ambassador
- Coordinate with other interviewers
  - ◇ know what to test on
  - ◇ look for patterns of mistakes
- Characteristics of a good problem:
  - ◇ no single point of failure
  - ◇ has multiple solutions
  - ◇ covers multiple areas
  - ◇ is calibrated on colleagues
  - ◇ does not require unnecessary domain knowledge
- Control the conversation
  - ◇ draw out quiet candidates
  - ◇ manage verbose/overconfident candidates
- Use a process for recording & scoring
- Determine what training is needed
- Apply the litmus test

(3.) Perform an initial phone screening, which often consists of a question-answer session over the phone or video chat with an engineer. You may be asked to submit code via a shared document or an online coding site such as ideone.com, collabedit.com, or coderpad.io. Don't take the screening casually—it can be extremely challenging.

(4.) Go for an on-site interview—this consists of a series of one-on-one interviews with engineers and managers, and a conversation with your Human Resources (HR) contact.

(5.) Receive offers—these are usually a starting point for negotiations.

Note that there may be variations—e.g., a company may contact you, or you may submit via your college's career placement center. The screening may involve a homework assignment to be done before or after the conversation. The on-site interview may be conducted over a video chat session. Most on-sites are half a day, but others may last the entire day. For anything involving interaction over a network, be absolutely sure to work out logistics (a quiet place to talk with a landline rather than a mobile, familiarity with the coding website and chat software, etc.) well in advance.

Table 1.2: First review Tables 1.3 on Page 10, 1.4 on Page 11, and 1.5 on Page 11. For each chapter, first read its introductory text. Use textbooks for reference only. Unless a problem is italicized, it entails writing code. For Scenario i, write and test code for the problems in Columns 0 to $i-1$, and pseudo-code for the problems in Column i.

Scenario 1 **Hackathon** **3 days**		**Scenario 2** **Finals cram** **7 days**	**Scenario 3** **Term project** **1 month**	**Scenario 4** **Algorithms class** **4 months**
C0	**C1**	**C2**	**C3**	**C4**
4.1	4.7	4.8	4.3, 4.11	4.9
5.1, 5.6	5.12, 5.18	5.2, 5.17	5.5, 5.9	5.3, 5.10, 5.15
6.1	6.2, 6.4	6.5, 6.6	6.7, 6.8	6.9, 6.11
7.1	7.2, 7.3	7.4, 7.7	7.10	7.11
8.1	8.6	8.2, 8.7	8.3, 8.8	8.4
9.1	9.4	9.2, 9.12	9.11	9.13, 9.16
10.1	10.4	10.3	10.5	10.6
11.1	11.4, 11.8	11.3, 11.9	11.5, 11.10	11.6, 11.7
12.2	12.3, 12.5	12.1, 12.5	12.4, 12.6	12.9
13.1	13.2	13.5	13.7, 13.10	13.8
14.1	14.2, 14.3	14.4	14.5, 14.8	14.7
15.1	15.2	15.3	15.4, 15.9	15.6, 15.10
16.1	16.2	16.3, 16.6	16.5, 16.7	16.12
17.4	17.6	17.5	17.7	17.8
18.1	18.7	18.2	18.3	18.5
19.3	19.6	*19.8*	*19.9*	*20.9*
20.13	*20.15*	*20.16*	*20.1*	*20.2*

We recommend that you interview at as many places as you can without it taking away from your job or classes. The experience will help you feel more comfortable with interviewing and you may discover you really like a company that you did not know much about.

The résumé

It always astonishes us to see candidates who've worked hard for at least four years in school, and often many more in the workplace, spend 30 minutes jotting down random factoids about themselves and calling the result a résumé.

A résumé needs to address HR staff, the individuals interviewing you, and the hiring manager. The HR staff, who typically first review your résumé, look for keywords, so you need to be sure you have those covered. The people interviewing you and the hiring manager need to know what you've done that makes you special, so you need to differentiate yourself.

Here are some key points to keep in mind when writing a résumé:

(1.) Have a clear statement of your objective; in particular, make sure that you tailor your résumé for a given employer. For example: My outstanding ability is developing solutions to computationally challenging problems; communicating them in written and oral form; and working with teams to implement them. I would like to apply these abilities at XYZ."

(2.) The most important points—the ones that differentiate you from everyone else—should come first. People reading your résumé proceed in sequential order, so you want to impress them with what makes you special early on. (Maintaining a logical flow, though desirable, is secondary compared to this principle.) As a consequence, you should not list your programming languages, coursework, etc. early on, since these are likely common to everyone. You should list significant class projects (this also helps with keywords for HR.), as well as talks/papers you've presented, and even standardized test scores, if truly exceptional.

(3.) The résumé should be of a high-quality: no spelling mistakes; consistent spacings, capitalizations, numberings; and correct grammar and punctuation. Use few fonts. Portable Document Format (PDF) is preferred, since it renders well across platforms.

(4.) Include contact information, a LinkedIn profile, and, ideally, a URL to a personal homepage with examples of your work. These samples may be class projects, a thesis, and links to companies and products you've worked on. Include design documents as well as a link to your version control repository.

(5.) If you can work at the company without requiring any special processing (e.g., if you have a Green Card, and are applying for a job in the US), make a note of that.

(6.) Have friends review your résumé; they are certain to find problems with it that you missed. It is better to get something written up quickly, and then refine it based on feedback.

(7.) A résumé does not have to be one page long—two pages are perfectly appropriate. (Over two pages is probably not a good idea.)

(8.) As a rule, we prefer not to see a list of hobbies/extracurricular activities (e.g., "reading books", "watching TV", "organizing tea party activities") unless they are really different (e.g., "Olympic rower") and not controversial.

Whenever possible, have a friend or professional acquaintance at the company route your résumé to the appropriate manager/HR contact—the odds of it reaching the right hands are much higher. At one company whose practices we are familiar with, a résumé submitted through a contact is 50 times more likely to result in a hire than one submitted online. Don't worry about wasting your contact's time—employees often receive a referral bonus, and being responsible for bringing in stars is also viewed positively.

Mock interviews

Mock interviews are a great way of preparing for an interview. Get a friend to ask you questions (from EPI or any other source) and solve them on a whiteboard, with pen and paper, or on a shared document. Have your friend take notes and give you feedback, both positive and negative. Make a video recording of the interview. You will cringe as you watch it, but it is better to learn of your mannerisms beforehand. Ask your friend to give hints when you get stuck. In addition to sharpening your problem solving and presentation skills, the experience will help reduce anxiety at the actual interview setting. If you cannot find a friend, you can still go through the same process, recording yourself.

Data structures, algorithms, and logic

We summarize the data structures, algorithms, and logical principles used in this book in Tables 1.3 on the next page, 1.4 on Page 11, and 1.5 on Page 11, and highly encourage you to review them. Don't be overly concerned if some of the concepts are new to you, as we will do a bootcamp review for data structures and algorithms at the start of the corresponding chapters. Logical principles are applied throughout the book, and we explain a principle in detail when we first use it. You can also look for the highlighted page in the index to learn more about a term.

Complexity

The run time of an algorithm depends on the size of its input. A common approach to capture the run time dependency is by expressing asymptotic bounds on the worst-case run time as a function

Table 1.3: Data structures

Data structure	Key points
Primitive types	Know how `int`, `char`, `double`, etc. are represented in memory and the primitive operations on them.
Arrays	Fast access for element at an index, slow lookups (unless sorted) and insertions. Be comfortable with notions of iteration, resizing, partitioning, merging, etc.
Strings	Know how strings are represented in memory. Understand basic operators such as comparison, copying, matching, joining, splitting, etc.
Lists	Understand trade-offs with respect to arrays. Be comfortable with iteration, insertion, and deletion within singly and doubly linked lists. Know how to implement a list with dynamic allocation, and with arrays.
Stacks and queues	Recognize where last-in first-out (stack) and first-in first-out (queue) semantics are applicable. Know array and linked list implementations.
Binary trees	Use for representing hierarchical data. Know about depth, height, leaves, search path, traversal sequences, successor/predecessor operations.
Heaps	Key benefit: $O(1)$ lookup find-max, $O(\log n)$ insertion, and $O(\log n)$ deletion of max. Node and array representations. Min-heap variant.
Hash tables	Key benefit: $O(1)$ insertions, deletions and lookups. Key disadvantages: not suitable for order-related queries; need for resizing; poor worst-case performance. Understand implementation using array of buckets and collision chains. Know hash functions for integers, strings, objects.
Binary search trees	Key benefit: $O(\log n)$ insertions, deletions, lookups, find-min, find-max, successor, predecessor when tree is height-balanced. Understand node fields, pointer implementation. Be familiar with notion of balance, and operations maintaining balance.

of the input size. Specifically, the run time of an algorithm on an input of size n is $O(f(n))$ if, for sufficiently large n, the run time is not more than $f(n)$ times a constant.

As an example, searching for a given integer in an unsorted array of integers of length n via iteration has an asymptotic complexity of $O(n)$ since in the worst-case, the given integer may not be present.

Complexity theory is applied in a similar manner when analyzing the space requirements of an algorithm. The space needed to read in an instance is not included; otherwise, every algorithm would have $O(n)$ space complexity. An algorithm that uses $O(1)$ space should not perform dynamic memory allocation (explicitly, or indirectly, e.g., through library routines). Furthermore, the maximum depth of the function call stack should also be a constant, independent of the input. The standard algorithm for depth-first search of a graph is an example of an algorithm that does

Table 1.4: Algorithms

Algorithm type	Key points
Sorting	Uncover some structure by sorting the input.
Recursion	If the structure of the input is defined in a recursive manner, design a recursive algorithm that follows the input definition.
Divide-and-conquer	Divide the problem into two or more smaller independent subproblems and solve the original problem using solutions to the subproblems.
Dynamic programming	Compute solutions for smaller instances of a given problem and use these solutions to construct a solution to the problem. Cache for performance.
Greedy algorithms	Compute a solution in stages, making choices that are locally optimum at each step; these choices are never undone.
Invariants	Identify an invariant and use it to rule out potential solutions that are suboptimal/dominated by other solutions.
Graph modeling	Describe the problem using a graph and solve it using an existing graph algorithm.

Table 1.5: Logical principles

Principle	Key points
Concrete examples	Manually solve concrete instances and then build a general solution. Try small inputs, e.g., a BST containing 5–7 elements, and extremal inputs, e.g., sorted arrays.
Case analysis	Split the input/execution into a number of cases and solve each case in isolation.
Iterative refinement	Most problems can be solved using a brute-force approach. Find such a solution and improve upon it.
Reduction	Use a known solution to some other problem as a subroutine.

not perform any dynamic allocation, but uses the function call stack for implicit storage—its space complexity is not $O(1)$.

A streaming algorithm is one in which the input is presented as a sequence of items and the algorithm makes a small number of passes over it (typically just one), using a limited amount memory (much less than the input size) and a limited processing time per item. The best algorithms for performing aggregation queries on log file data are often streaming algorithms.

Language review

Programs are written and tested in Python 3.6. Most of them will work with earlier versions of Python as well. Some of the newer language features we use are `concurrent.futures` for thread pools, and the `ABC` for abstract base classes. The only external dependency we have is on `bintrees`, which implements a balanced binary search tree (Chapter 14).

We review data structures in Python in the corresponding chapters. Here we describe some Python language features that go beyond the basics that we find broadly applicable. Be sure you are comfortable with the ideas behind these features, as well as their basic syntax and time complexity.

- We use inner functions and lambdas, e.g., the sorting code on Page 181. You should be especially be comfortable with lambda syntax, as well as the variable scoping rules for inner functions and lambdas.
- We use `collections.namedtuples` extensively for structured data—these are more readable than dictionaries, lists, and tuples, and less verbose than classes.
- We use the following constructs to write simpler code: `all()` and `any()`, list comprehension, `map()`, `functools.reduce()` and `zip()`, and `enumerate()`.
- The following functions from the `itertools` module are very useful in diverse contexts: `groupby()`, `accumulate()`, `product()`, and `combinations()`.

For a handful of problems, when presenting their solution, we also include a Pythonic solution. This is indicated by the use of `_pythonic` for the suffix of the function name. These Pythonic programs are not solutions that interviewers would expect of you—they are supposed to fill you with a sense of joy and wonder. (If you find Pythonic solutions to problems, please share them with us!)

Best practices for interview code

Now we describe practices we use in EPI that are not suitable for production code. They are necessitated by the finite time constraints of an interview. See Section 2 on Page 14 for more insights.

- We make fields public, rather than use getters and setters.
- We do not protect against invalid inputs, e.g., null references, negative entries in an array that's supposed to be all nonnegative, input streams that contain objects that are not of the expected type, etc.

Now we describe practices we follow in EPI which are industry-standard, but we would not recommend for an interview.

- We follow the PEP 8 style guide, which you should review before diving into EPI. The guide is fairly straightforward—it mostly addresses naming and spacing conventions, which should not be a high priority when presenting your solution.

An industry best practice that we use in EPI and recommend you use in an interview is explicitly creating classes for data clumps, i.e., groups of values that do not have any methods on them. Many programmers would use a generic Pair or Tuple class, but we have found that this leads to confusing and buggy programs.

Books

Our favorite introduction to Python is Severance's "Python for Informatics: Exploring Information", which does a great job of covering the language constructs with examples. It is available for free online.

Brett Slatkin's "Effective Python" is one of the best all-round programming books we have come across, addressing everything from the pitfalls of default arguments to concurrency patterns.

For design patterns, we like "Head First Design Patterns" by Freeman *et al.*. Its primary drawback is its bulk. Note that programs for interviews are too short to take advantage of design patterns.

CHAPTER

2 Strategies For A Great Interview

The essence of strategy is choosing what not to do.

— M. E. PORTER

The technical component of an onsite interview usually consists of three to five one-on-one interviews with engineers. A typical one hour interview with a single interviewer consists of five minutes of introductions and questions about the candidate's résumé. This is followed by five to fifteen minutes of questioning on basic programming concepts. The core of the interview is one or two problems where the candidate is expected to present a detailed solution on a whiteboard, paper, or integrated development environments (IDEs). Depending on the interviewer and the question, the solution may be required to include syntactically correct code and tests. Junior candidates should expect more emphasis on coding, and less on system design and architecture. Senior candidates should expect more emphasis on system design and architecture, though at least one interviewer will ask a problem that entails writing detailed code.

Approaching the problem

No matter how clever and well prepared you are, the solution to an interview problem may not occur to you immediately. Here are some things to keep in mind when this happens.

Clarify the question: This may seem obvious but it is amazing how many interviews go badly because the candidate spends most of his time trying to solve the wrong problem. If a question seems exceptionally hard, you may have misunderstood it.

A good way of clarifying the question is to state a concrete instance of the problem. For example, if the question is "find the first occurrence of a number greater than k in a sorted array", you could ask "if the input array is $\langle 2, 20, 30 \rangle$ and k is 3, then are you supposed to return 1, the index of 20?" These questions can be formalized as unit tests.

Feel free to ask the interviewer what time and space complexity he would like in your solution. If you are told to implement an $O(n)$ algorithm or use $O(1)$ space, it can simplify your life considerably. It is possible that he will refuse to specify these, or be vague about complexity requirements, but there is no harm in asking. Even if they are evasive, you may get some clues.

Work on concrete examples: Consider the problem of determining the smallest amount of change that you cannot make with a given set of coins, as described on Page 186. This problem may seem difficult at first. However, if you try out the smallest amount that cannot be made with some small examples, e.g., $\{1, 2\}$, $\{1, 3\}$, $\{1, 2, 4\}$, $\{1, 2, 5\}$, you will get the key insights: examine coins in sorted order, and look for a large "jump"—a coin which is larger than the sum of the preceding coins.

Spell out the brute-force solution: Problems that are put to you in an interview tend to have an obvious brute-force solution that has a high time complexity compared to more sophisticated solutions. For example, instead of trying to work out a DP solution for a problem (e.g., for Problem 16.7 on Page 249), try all the possible configurations. Advantages to this approach include: (1.) it helps you explore opportunities for optimization and hence reach a better solution, (2.) it gives you an opportunity to demonstrate some problem solving and coding skills, and (3.) it establishes that both you and the interviewer are thinking about the same problem. Be warned that this strategy can sometimes be detrimental if it takes a long time to describe the brute-force approach.

Think out loud: One of the worst things you can do in an interview is to freeze up when solving the problem. It is always a good idea to think out loud and stay engaged. On the one hand, this increases your chances of finding the right solution because it forces you to put your thoughts in a coherent manner. On the other hand, this helps the interviewer guide your thought process in the right direction. Even if you are not able to reach the solution, the interviewer will form some impression of your intellectual ability.

Apply patterns: Patterns—general reusable solutions to commonly occurring problems—can be a good way to approach a baffling problem. Examples include finding a good data structure, seeing if your problem is a good fit for a general algorithmic technique, e.g., divide-and-conquer, recursion, or dynamic programming, and mapping the problem to a graph.

Presenting the solution

Once you have an algorithm, it is important to present it in a clear manner. Your solution will be much simpler if you take advantage of libraries such as Java Collections, C++ STL, or Python collections. However, it is far more important that you use the language you are most comfortable with. Here are some things to keep in mind when presenting a solution.

Libraries: Do not reinvent the wheel (unless asked to invent it). In particular, master the libraries, especially the data structures. For example, do not waste time and lose credibility trying to remember how to pass an explicit comparator to a BST constructor. Remember that a hash function should use exactly those fields which are used in the equality check. A comparison function should be transitive.

Focus on the top-level algorithm: It's OK to use functions that you will implement later. This will let you focus on the main part of the algorithm, will penalize you less if you don't complete the algorithm. (Hash, equals, and compare functions are good candidates for deferred implementation.) Specify that you will handle main algorithm first, then corner cases. Add TODO comments for portions that you want to come back to.

Manage the whiteboard: You will likely use more of the board than you expect, so start at the top-left corner. Make use of functions—skip implementing anything that's trivial (e.g., finding the maximum of an array) or standard (e.g., a thread pool). Best practices for coding on a whiteboard are very different from best practices for coding on a production project. For example, don't worry about skipping documentation, or using the right indentation. Writing on a whiteboard is much slower than on a keyboard, so keeping your identifiers short (our recommendation is no more than 7 characters) but recognizable is a best practice. Have a convention for identifiers, e.g., `i,j,k` for array indices, `A,B,C` for arrays, `s` for `string`, `d` for `dict`, etc.

Assume valid inputs: In a production environment, it is good practice to check if inputs are valid, e.g., that a string purporting to represent a nonnegative integer actually consists solely of numeric characters, no flight in a timetable arrives before it departs, etc. Unless they are part of the problem statement, in an interview setting, such checks are inappropriate: they take time to code, and distract from the core problem. (You should clarify this assumption with the interviewer.)

Test for corner cases: For many problems, your general idea may work for most valid inputs but there may be pathological valid inputs where your algorithm (or your implementation of it) fails. For example, your binary search code may crash if the input is an empty array; or you may do arithmetic without considering the possibility of overflow. It is important to systematically consider these possibilities. If there is time, write unit tests. Small, extreme, or random inputs make for good stimuli. Don't forget to add code for checking the result. Occasionally, the code to handle obscure corner cases may be too complicated to implement in an interview setting. If so, you should mention to the interviewer that you are aware of these problems, and could address them if required.

Syntax: Interviewers rarely penalize you for small syntax errors since modern IDE excel at handling these details. However, lots of bad syntax may result in the impression that you have limited coding experience. Once you are done writing your program, make a pass through it to fix any obvious syntax errors before claiming you are done.

Candidates often tend to get function signatures wrong and it reflects poorly on them. For example, it would be an error to write a function in C that returns an array but not its size.

Memory management: Generally speaking, it is best to avoid memory management operations altogether. See if you can reuse space. For example, some linked list problems can be solved with $O(1)$ additional space by reusing existing nodes.

Your Interviewer Is Not Alan Turing: Interviewers are not capable of analyzing long programs, particularly on a whiteboard or paper. Therefore, they ask questions whose solutions use short programs. A good tip is that if your solution takes more than 50 lines to code in Python, it's a sign that you are on the wrong track, and you should reconsider your approach.

Know your interviewers & the company

It can help you a great deal if the company can share with you the background of your interviewers in advance. You should use search and social networks to learn more about the people interviewing you. Letting your interviewers know that you have researched them helps break the ice and forms the impression that you are enthusiastic and will go the extra mile. For fresh graduates, it is also important to think from the perspective of the interviewers as described in Chapter 3.

Once you ace your interviews and have an offer, you have an important decision to make—is this the organization where you want to work? Interviews are a great time to collect this information. Interviews usually end with the interviewers letting the candidates ask questions. You should make the best use of this time by getting the information you would need and communicating to the interviewer that you are genuinely interested in the job. Based on your interaction with the interviewers, you may get a good idea of their intellect, passion, and fairness. This extends to the team and company.

In addition to knowing your interviewers, you should know about the company vision, history, organization, products, and technology. You should be ready to talk about what specifically appeals

to you, and to ask intelligent questions about the company and the job. Prepare a list of questions in advance; it gets you helpful information as well as shows your knowledge and enthusiasm for the organization. You may also want to think of some concrete ideas around things you could do for the company; be careful not to come across as a pushy know-it-all.

All companies want bright and motivated engineers. However, companies differ greatly in their culture and organization. Here is a brief classification.

Mature consumer-facing company, e.g., Google: wants candidates who understand emerging technologies from the user's perspective. Such companies have a deeper technology stack, much of which is developed in-house. They have the resources and the time to train a new hire.

Enterprise-oriented company, e.g., Oracle: looks for developers familiar with how large projects are organized, e.g., engineers who are familiar with reviews, documentation, and rigorous testing.

Government contractor, e.g., Lockheed-Martin: values knowledge of specifications and testing, and looks for engineers who are familiar with government-mandated processes.

Startup, e.g., Uber: values engineers who take initiative and develop products on their own. Such companies do not have time to train new hires, and tend to hire candidates who are very fast learners or are already familiar with their technology stack, e.g., their web application framework, machine learning system, etc.

Embedded systems/chip design company, e.g., National Instruments: wants software engineers who know enough about hardware to interface with the hardware engineers. The tool chain and development practices at such companies tend to be very mature.

General conversation

Often interviewers will ask you questions about your past projects, such as a senior design project or an internship. The point of this conversation is to answer the following questions:

Can the candidate clearly communicate a complex idea? This is one of the most important skills for working in an engineering team. If you have a grand idea to redesign a big system, can you communicate it to your colleagues and bring them on board? It is crucial to practice how you will present your best work. Being precise, clear, and having concrete examples can go a long way here. Candidates communicating in a language that is not their first language, should take extra care to speak slowly and make more use of the whiteboard to augment their words.

Is the candidate passionate about his work? We always want our colleagues to be excited, energetic, and inspiring to work with. If you feel passionately about your work, and your eyes light up when describing what you've done, it goes a long way in establishing you as a great colleague. Hence, when you are asked to describe a project from the past, it is best to pick something that you are passionate about rather than a project that was complex but did not interest you.

Is there a potential interest match with some project? The interviewer may gauge areas of strengths for a potential project match. If you know the requirements of the job, you may want to steer the conversation in that direction. Keep in mind that because technology changes so fast many teams prefer a strong generalist, so don't pigeonhole yourself.

Other advice

A bad mental and physical attitude can lead to a negative outcome. Don't let these simple mistakes lead to your years of preparation going to waste.

Be honest: Nobody wants a colleague who falsely claims to have tested code or done a code review. Dishonesty in an interview is a fast pass to an early exit.

Remember, nothing breaks the truth more than stretching it—you should be ready to defend anything you claim on your résumé. If your knowledge of Python extends only as far as having cut-and-paste sample code, do not add Python to your résumé.

Similarly, if you have seen a problem before, you should say so. (Be sure that it really is the same problem, and bear in mind you should describe a correct solution quickly if you claim to have solved it before.) Interviewers have been known to collude to ask the same question of a candidate to see if he tells the second interviewer about the first instance. An interviewer may feign ignorance on a topic he knows in depth to see if a candidate pretends to know it.

Keep a positive spirit: A cheerful and optimistic attitude can go a long way. Absolutely nothing is to be gained, and much can be lost, by complaining how difficult your journey was, how you are not a morning person, how inconsiderate the airline/hotel/HR staff were, etc.

Don't apologize: Candidates sometimes apologize in advance for a weak GPA, rusty coding skills, or not knowing the technology stack. Their logic is that by being proactive they will somehow benefit from lowered expectations. Nothing can be further from the truth. It focuses attention on shortcomings. More generally, if you do not believe in yourself, you cannot expect others to believe in you.

Keep money and perks out of the interview: Money is a big element in any job but it is best left discussed with the HR division after an offer is made. The same is true for vacation time, day care support, and funding for conference travel.

Appearance: Most software companies have a relaxed dress-code, and new graduates may wonder if they will look foolish by overdressing. The damage done when you are too casual is greater than the minor embarrassment you may feel at being overdressed. It is always a good idea to err on the side of caution and dress formally for your interviews. At the minimum, be clean and well-groomed.

Be aware of your body language: Think of a friend or coworker slouched all the time or absentmindedly doing things that may offend others. Work on your posture, eye contact and handshake, and remember to smile.

Stress interviews

Some companies, primarily in the finance industry, make a practice of having one of the interviewers create a stressful situation for the candidate. The stress may be injected technically, e.g., via a ninja problem, or through behavioral means, e.g., the interviewer rejecting a correct answer or ridiculing the candidate. The goal is to see how a candidate reacts to such situations—does he fall apart, become belligerent, or get swayed easily. The guidelines in the previous section should help you through a stress interview. (Bear in mind you will not know *a priori* if a particular interviewer will be conducting a stress interview.)

Learning from bad outcomes

The reality is that not every interview results in a job offer. There are many reasons for not getting a particular job. Some are technical: you may have missed that key flash of insight, e.g., the key

to solving the maximum-profit on Page 1 in linear time. If this is the case, go back and solve that problem, as well as related problems.

Often, your interviewer may have spent a few minutes looking at your résumé—this is a depressingly common practice. This can lead to your being asked questions on topics outside of the area of expertise you claimed on your résumé, e.g., routing protocols or Structured Query Language (SQL). If so, make sure your résumé is accurate, and brush up on that topic for the future.

You can fail an interview for nontechnical reasons, e.g., you came across as uninterested, or you did not communicate clearly. The company may have decided not to hire in your area, or another candidate with similar ability but more relevant experience was hired.

You will not get any feedback from a bad outcome, so it is your responsibility to try and piece together the causes. Remember the only mistakes are the ones you don't learn from.

Negotiating an offer

An offer is not an offer till it is on paper, with all the details filled in. All offers are negotiable. We have seen compensation packages bargained up to twice the initial offer, but 10–20% is more typical. When negotiating, remember there is nothing to be gained, and much to lose, by being rude. (Being firm is not the same as being rude.)

To get the best possible offer, get multiple offers, and be flexible about the form of your compensation. For example, base salary is less flexible than stock options, sign-on bonus, relocation expenses, and Immigration and Naturalization Service (INS) filing costs. Be concrete—instead of just asking for more money, ask for a P% higher salary. Otherwise the recruiter will simply come back with a small increase in the sign-on bonus and claim to have met your request.

Your HR contact is a professional negotiator, whose fiduciary duty is to the company. He will know and use negotiating techniques such as reciprocity, getting consensus, putting words in your mouth ("don't you think that's reasonable?"), as well as threats, to get the best possible deal for the company. (This is what recruiters themselves are evaluated on internally.) The Wikipedia article on negotiation lays bare many tricks we have seen recruiters employ.

One suggestion: stick to email, where it is harder for someone to paint you into a corner. If you are asked for something (such as a copy of a competing offer), get something in return. Often it is better to bypass the HR contact and speak directly with the hiring manager.

At the end of the day, remember your long term career is what counts, and joining a company that has a brighter future (social-mobile vs. legacy enterprise), or offers a position that has more opportunities to rise (developer vs. tester) is much more important than a 10–20% difference in compensation.

CHAPTER

3 Conducting An Interview

知己知彼，百戰不殆。
Translated—"If you know both yourself and your enemy, you can win numerous battles without jeopardy."

— *"The Art of War,"*
SUN TZU, 515 B.C.

In this chapter we review practices that help interviewers identify a top hire. We strongly recommend interviewees read it—knowing what an interviewer is looking for will help you present yourself better and increase the likelihood of a successful outcome.

For someone at the beginning of their career, interviewing may feel like a huge responsibility. Hiring a bad candidate is expensive for the organization, not just because the hire is unproductive, but also because he is a drain on the productivity of his mentors and managers, and sets a bad example. Firing someone is extremely painful as well as bad for to the morale of the team. On the other hand, discarding good candidates is problematic for a rapidly growing organization. Interviewers also have a moral responsibility not to unfairly crush the interviewee's dreams and aspirations.

Objective

The ultimate goal of any interview is to determine the odds that a candidate will be a successful employee of the company. The ideal candidate is smart, dedicated, articulate, collegial, and gets things done quickly, both as an individual and in a team. Ideally, your interviews should be designed such that a good candidate scores 1.0 and a bad candidate scores 0.0.

One mistake, frequently made by novice interviewers, is to be indecisive. Unless the candidate walks on water or completely disappoints, the interviewer tries not to make a decision and scores the candidate somewhere in the middle. This means that the interview was a wasted effort.

A secondary objective of the interview process is to turn the candidate into a brand ambassador for the recruiting organization. Even if a candidate is not a good fit for the organization, he may know others who would be. It is important for the candidate to have an overall positive experience during the process. It seems obvious that it is a bad idea for an interviewer to check email while the candidate is talking or insult the candidate over a mistake he made, but such behavior is depressingly common. Outside of a stress interview, the interviewer should work on making the candidate feel positively about the experience, and, by extension, the position and the company.

What to ask

One important question you should ask yourself as an interviewer is how much training time your work environment allows. For a startup it is important that a new hire is productive from the first week, whereas a larger organization can budget for several months of training. Consequently, in a startup it is important to test the candidate on the specific technologies that he will use, in addition to his general abilities.

For a larger organization, it is reasonable not to emphasize domain knowledge and instead test candidates on data structures, algorithms, system design skills, and problem solving techniques. The justification for this is as follows. Algorithms, data structures, and system design underlie all software. Algorithms and data structure code is usually a small component of a system dominated by the user interface (UI), input/output (I/O), and format conversion. It is often hidden in library calls. However, such code is usually the crucial component in terms of performance and correctness, and often serves to differentiate products. Furthermore, platforms and programming languages change quickly but a firm grasp of data structures, algorithms, and system design principles, will always be a foundational part of any successful software endeavor. Finally, many of the most successful software companies have hired based on ability and potential rather than experience or knowledge of specifics, underlying the effectiveness of this approach to selecting candidates.

Most big organizations have a structured interview process where designated interviewers are responsible for probing specific areas. For example, you may be asked to evaluate the candidate on their coding skills, algorithm knowledge, critical thinking, or the ability to design complex systems. This book gives interviewers access to a fairly large collection of problems to choose from. When selecting a problem keep the following in mind:

No single point of failure—if you are going to ask just one question, you should not pick a problem where the candidate passes the interview if and only if he gets one particular insight. The best candidate may miss a simple insight, and a mediocre candidate may stumble across the right idea. There should be at least two or three opportunities for the candidates to redeem themselves. For example, problems that can be solved by dynamic programming can almost always be solved through a greedy algorithm that is fast but suboptimum or a brute-force algorithm that is slow but optimum. In such cases, even if the candidate cannot get the key insight, he can still demonstrate some problem solving abilities. Problem 5.6 on Page 46 exemplifies this type of question.

Multiple possible solutions—if a given problem has multiple solutions, the chances of a good candidate coming up with a solution increases. It also gives the interviewer more freedom to steer the candidate. A great candidate may finish with one solution quickly enough to discuss other approaches and the trade-offs between them. For example, Problem 11.9 on Page 155 can be solved using a hash table or a bit array; the best solution makes use of binary search.

Cover multiple areas—even if you are responsible for testing the candidate on algorithms, you could easily pick a problem that also exposes some aspects of design and software development. For example, Problem 19.8 on Page 298 tests candidates on concurrency as well as data structures. Problem 5.16 on Page 58 requires knowledge of both probability and binary search.

Calibrate on colleagues—interviewers often have an incorrect notion of how difficult a problem is for a thirty minute or one hour interview. It is a good idea to check the appropriateness of a problem by asking one of your colleagues to solve it and seeing how much difficulty they have with

it.

No unnecessary domain knowledge—it is not a good idea to quiz a candidate on advanced graph algorithms if the job does not require it and the candidate does not claim any special knowledge of the field. (The exception to this rule is if you want to test the candidate's response to stress.)

Conducting the interview

Conducting a good interview is akin to juggling. At a high level, you want to ask your questions and evaluate the candidate's responses. Many things can happen in an interview that could help you reach a decision, so it is important to take notes. At the same time, it is important to keep a conversation going with the candidate and help him out if he gets stuck. Ideally, have a series of hints worked out beforehand, which can then be provided progressively as needed. Coming up with the right set of hints may require some thinking. You do not want to give away the problem, yet find a way for the candidate to make progress. Here are situations that may throw you off:

A candidate that gets stuck and shuts up: Some candidates get intimidated by the problem, the process, or the interviewer, and just shut up. In such situations, a candidate's performance does not reflect his true caliber. It is important to put the candidate at ease, e.g., by beginning with a straightforward question, mentioning that a problem is tough, or asking them to think out loud.

A verbose candidate: Candidates who go off on tangents and keep on talking without making progress render an interview ineffective. Again, it is important to take control of the conversation. For example you could assert that a particular path will not make progress.

An overconfident candidate: It is common to meet candidates who weaken their case by defending an incorrect answer. To give the candidate a fair chance, it is important to demonstrate to him that he is making a mistake, and allow him to correct it. Often the best way of doing this is to construct a test case where the candidate's solution breaks down.

Scoring and reporting

At the end of an interview, the interviewers usually have a good idea of how the candidate scored. However, it is important to keep notes and revisit them before making a final decision. Whiteboard snapshots and samples of any code that the candidate wrote should also be recorded. You should standardize scoring based on which hints were given, how many questions the candidate was able to get to, etc. Although isolated minor mistakes can be ignored, sometimes when you look at all the mistakes together, clear signs of weakness in certain areas may emerge, such as a lack of attention to detail and unfamiliarity with a language.

When the right choice is not clear, wait for the next candidate instead of possibly making a bad hiring decision. The litmus test is to see if you would react positively to the candidate replacing a valuable member of your team.

Part II

Data Structures and Algorithms

CHAPTER

4 Primitive Types

Representation is the essence of programming.

— "The Mythical Man Month,"
F. P. BROOKS, 1975

A program updates variables in memory according to its instructions. Variables come in types—a type is a classification of data that spells out possible values for that type and the operations that can be performed on it. A type can be provided by the language or defined by the programmer. In Python everything is an object—this includes Booleans, integers, characters, etc.

Primitive types boot camp

Writing a program to count the number of bits that are set to 1 in a positive integer is a good way to get up to speed with primitive types. The following program tests bits one-at-a-time starting with the least-significant bit. It illustrates shifting and masking; it also shows how to avoid hard-coding the size of the integer word.

```
def count_bits(x):
    num_bits = 0
    while x:
        num_bits += x & 1
        x >>= 1
    return num_bits
```

Since we perform $O(1)$ computation per bit, the time complexity is $O(n)$, where n is the number of bits needed to represent the integer, e.g., 4 bits are needed to represent 12, which is $(1100)_2$ in binary. The techniques in Solution 4.1 on the next page, which is concerned with counting the number of bits mod 2, i.e., the parity, can be used to improve the performance of the program given above.

Know your primitive types

Python has a number of built-in types: numerics (e.g., integer), sequences (e.g., list), mappings (e.g., dict), as well as classes, instances and exceptions. All instances of these types are objects.

Integers in Python3 are unbounded—the maximum integer representable is a function of the available memory. The constant `sys.maxsize` can be used to find the word-size; specifically, it's the maximum value integer that can be stored in the word, e.g., `2**63 - 1` on a 64-bit machine. Bounds on floats are specified in `sys.float_info`.

- Be very familiar with the bit-wise operators such as `6&4,1|2, 8>>1, -16>>2, 1<<10, ~0, 15^x`. Negative numbers are treated as their 2's complement value. (There is no concept of an unsigned shift in Python, since integers have infinite precision.)

Be very comfortable with the **bitwise operators**, particularly XOR.
Understand how to use **masks** and create them in an **machine independent** way.
Know fast ways to **clear the lowermost set bit** (and by extension, set the lowermost 0, get its index, etc.)
Understand **signedness** and its implications to **shifting**.
Consider using a **cache** to accelerate operations by using it to brute-force small inputs.
Be aware that **commutativity** and **associativity** can be used to perform operations in **parallel** and **reorder** operations.

Table 4.1: Top Tips for Primitive Types

- The key methods for numeric types are `abs(-34.5)`, `math.ceil(2.17)`, `math.floor(3.14)`, `min(x,-4)`, `max(3.14, y)`, `pow(2.71, 3.14)` (alternately, `2.71 ** 3.14`), and `math.sqrt(225)`.
- Know how to interconvert integers and strings, e.g., `str(42)`, `int('42')`, floats and strings, e.g., `str(3.14)`, `float('3.14')`.
- Unlike integers, floats are not infinite precision, and it's convenient to refer to infinity as `float('inf')` and `float('-inf')`. These values are comparable to integers, and can be used to create psuedo max-int and pseudo min-int.
- When comparing floating point values consider using `math.isclose()`—it is robust, e.g., when comparing very large values, and can handle both absolute and relative differences.
- The key methods in `random` are `random.randrange(28)`, `random.randint(8,16)`, `random.random()`, `random.shuffle(A)`, and `random.choice(A)`.

4.1 Computing the parity of a word

The parity of a binary word is 1 if the number of 1s in the word is odd; otherwise, it is 0. For example, the parity of 1011 is 1, and the parity of 10001000 is 0. Parity checks are used to detect single bit errors in data storage and communication. It is fairly straightforward to write code that computes the parity of a single 64-bit word.

How would you compute the parity of a very large number of 64-bit words?

Hint: Use a lookup table, but don't use 2^{64} entries!

Solution: The brute-force algorithm iteratively tests the value of each bit while tracking the number of 1s seen so far. Since we only care if the number of 1s is even or odd, we can store the number mod 2.

```
def parity(x):
    result = 0
    while x:
        result ^= x & 1
```

```
        x >>= 1
    return result
```

The time complexity is $O(n)$, where n is the word size.

We will now describe several algorithms for parity computation that are superior to the brute-force algorithm.

The first improvement is based on erasing the lowest set bit in a word in a single operation, thereby improving performance in the best- and average-cases. Here is a great bit-fiddling trick which you should commit to memory: $x \,\&(x-1)$ equals x with its lowest set bit erased. (Here & is the bitwise-AND operator.) [1] For example, if $x = (00101100)_2$, then $x - 1 = (00101011)_2$, so $x \,\&(x-1) = (00101100)_2 \& (00101011)_2 = (00101000)_2$. This fact can be used to reduce the time complexity. Let k be the number of bits set to 1 in a particular word. (For example, for 10001010, $k = 3$.) Then time complexity of the algorithm below is $O(k)$.

```
def parity(x):
    result = 0
    while x:
        result ^= 1
        x &= x - 1  # Drops the lowest set bit of x.
    return result
```

We now consider a qualitatively different approach. The problem statement refers to computing the parity for a very large number of words. When you have to perform a large number of parity computations, and, more generally, any kind of bit fiddling computations, two keys to performance are processing multiple bits at a time and caching results in an array-based lookup table.

First we demonstrate caching. Clearly, we cannot cache the parity of every 64-bit integer—we would need 2^{64} bits of storage, which is of the order of two exabytes. However, when computing the parity of a collection of bits, it does not matter how we group those bits, i.e., the computation is associative. Therefore, we can compute the parity of a 64-bit integer by grouping its bits into four nonoverlapping 16 bit subwords, computing the parity of each subwords, and then computing the parity of these four subresults. We choose 16 since $2^{16} = 65536$ is relatively small, which makes it feasible to cache the parity of all 16-bit words using an array. Furthermore, since 16 evenly divides 64, the code is simpler than if we were, for example, to use 10 bit subwords.

We illustrate the approach with a lookup table for 2-bit words. The cache is $\langle 0, 1, 1, 0 \rangle$—these are the parities of (00), (01), (10), (11), respectively. To compute the parity of (11001010) we would compute the parities of (11), (00), (10), (10). By table lookup we see these are $0, 0, 1, 1$, respectively, so the final result is the parity of $0, 0, 1, 1$ which is 0.

To lookup the parity of the first two bits in (11101010), we right shift by 6, to get (00000011), and use this as an index into the cache. To lookup the parity of the next two bits, i.e., (10), we right shift by 4, to get (10) in the two least-significant bit places. The right shift does not remove the leading (11)—it results in (00001110). We cannot index the cache with this, it leads to an out-of-bounds access. To get the last two bits after the right shift by 4, we bitwise-AND (00001110) with (00000011) (this is the "mask" used to extract the last 2 bits). The result is (00000010). Similar masking is needed for the two other 2-bit lookups.

[1] Another key bit-fiddling trick: $x \,\&\sim(x-1)$ isolates the lowest bit that is 1 in x; here $\sim$ is the bitwise complement operator.

```
def parity(x):
    MASK_SIZE = 16
    BIT_MASK = 0xFFFF
    return (PRECOMPUTED_PARITY[x >> (3 * MASK_SIZE)] ^
            PRECOMPUTED_PARITY[(x >> (2 * MASK_SIZE)) & BIT_MASK] ^
            PRECOMPUTED_PARITY[(x >> MASK_SIZE)
                               & BIT_MASK] ^ PRECOMPUTED_PARITY[x & BIT_MASK])
```

The time complexity is a function of the size of the keys used to index the lookup table. Let L be the width of the words for which we cache the results, and n the word size. Since there are n/L terms, the time complexity is $O(n/L)$, assuming word-level operations, such as shifting, take $O(1)$ time. (This does not include the time for initialization of the lookup table.)

We can improve on the $O(n)$ worst-case time complexity of the previous algorithms by exploiting some simple properties of parity. Specifically, the XOR of two bits is defined to be 0 if both bits are 0 or both bits are 1; otherwise it is 1. XOR has the property of being associative, i.e., it does not matter how we group bits, as well as commutative, i.e., the order in which we perform the XORs does not change the result. The XOR of a group of bits is its parity. We can exploit this fact to use the CPU's word-level XOR instruction to process multiple bits at a time.

For example, the parity of $\langle b_{63}, b_{62}, \ldots, b_3, b_2, b_1, b_0 \rangle$ equals the parity of the XOR of $\langle b_{63}, b_{62}, \ldots, b_{32} \rangle$ and $\langle b_{31}, b_{30}, \ldots, b_0 \rangle$. The XOR of these two 32-bit values can be computed with a single shift and a single 32-bit XOR instruction. We repeat the same operation on 32-, 16-, 8-, 4-, 2-, and 1-bit operands to get the final result. Note that the leading bits are not meaningful, and we have to explicitly extract the result from the least-significant bit.

We illustrate the approach with an 8-bit word. The parity of (11010111) is the same as the parity of (1101) XORed with (0111), i.e., of (1010). This in turn is the same as the parity of (10) XORed with (10), i.e., of (00). The final result is the XOR of (0) with (0), i.e., 0. Note that the first XOR yields (11011010), and only the last 4 bits are relevant going forward. The second XOR yields (11101100), and only the last 2 bits are relevant. The third XOR yields (10011010). The last bit is the result, and to extract it we have to bitwise-AND with (00000001).

```
def parity(x):
    x ^= x >> 32
    x ^= x >> 16
    x ^= x >> 8
    x ^= x >> 4
    x ^= x >> 2
    x ^= x >> 1
    return x & 0x1
```

The time complexity is $O(\log n)$, where n is the word size.

Note that we can combine caching with word-level operations, e.g., by doing a lookup in the XOR-based approach once we get to 16 bits.

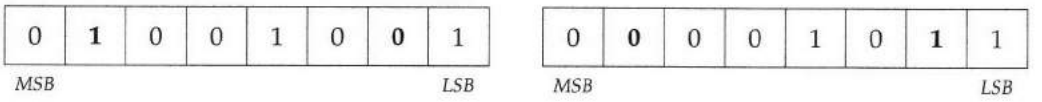
The actual runtimes depend on the input data, e.g., the refinement of the brute-force algorithm is very fast on sparse inputs. However, for random inputs, the refinement of the brute-force is roughly 20% faster than the brute-force algorithm. The table-based approach is four times faster still, and using associativity reduces run time by another factor of two.

Variant: Write expressions that use bitwise operators, equality checks, and Boolean operators to do the following in $O(1)$ time.

- Right propagate the rightmost set bit in x, e.g., turns $(01010000)_2$ to $(01011111)_2$.
- Compute x mod a power of two, e.g., returns 13 for 77 mod 64.
- Test if x is a power of 2, i.e., evaluates to true for $x = 1, 2, 4, 8, \ldots$, false for all other values.

4.2 Swap bits

There are a number of ways in which bit manipulations can be accelerated. For example, as described on Page 23, the expression x & $(x-1)$ clears the lowest set bit in x, and x & $\sim(x-1)$ extracts the lowest set bit of x. Here are a few examples: $16\&(16-1) = 0$, $11\&(11-1) = 10$, $20\&(20-1) = 16$, $16\&\sim(16-1) = 16$, $11\&\sim(11-1) = 1$, and $20\&\sim(20-1) = 4$.



(a) The 8-bit integer 73 can be viewed as array of bits, with the LSB being at index 0.

(b) The result of swapping the bits at indices 1 and 6, with the LSB being at index 0. The corresponding integer is 11.

Figure 4.1: Example of swapping a pair of bits.

A 64-bit integer can be viewed as an array of 64 bits, with the bit at index 0 corresponding to the least significant bit (LSB), and the bit at index 63 corresponding to the most significant bit (MSB). Implement code that takes as input a 64-bit integer and swaps the bits at indices i and j. Figure 4.1 illustrates bit swapping for an 8-bit integer.

Hint: When is the swap necessary?

Solution: A brute-force approach would be to use bitmasks to extract the ith and jth bits, saving them to local variables. Consequently, write the saved jth bit to index i and the saved ith bit to index j, using a combination of bitmasks and bitwise operations.

The brute-force approach works generally, e.g., if we were swapping objects stored in an array. However, since a bit can only take two values, we can do a little better. Specifically, we first test if the bits to be swapped differ. If they do not, the swap does not change the integer. If the bits are different, swapping them is the same as flipping their individual values. For example in Figure 4.1, since the bits at Index 1 and Index 6 differ, flipping each bit has the effect of a swap.

In the code below we use standard bit-fiddling idioms for testing and flipping bits. Overall, the resulting code is slightly more succinct and efficient than the brute force approach.

```
def swap_bits(x, i, j):
    # Extract the i-th and j-th bits, and see if they differ.
    if (x >> i) & 1 != (x >> j) & 1:
        # i-th and j-th bits differ. We will swap them by flipping their values.
        # Select the bits to flip with bit_mask. Since x^1 = 0 when x = 1 and 1
        # when x = 0, we can perform the flip XOR.
        bit_mask = (1 << i) | (1 << j)
        x ^= bit_mask
    return x
```

The time complexity is $O(1)$, independent of the word size.

4.3 Reverse bits

Write a program that takes a 64-bit unsigned integer and returns the 64-bit unsigned integer consisting of the bits of the input in reverse order. For example, if the input is (1110000000000001), the output should be (1000000000000111).

Hint: Use a lookup table.

Solution: If we need to perform this operation just once, there is a simple brute-force algorithm: iterate through the 32 least significant bits of the input, and swap each with the corresponding most significant bit, using, for example, the approach in Solution 4.2 on the preceding page.

To implement reverse when the operation is to be performed repeatedly, we look more carefully at the structure of the input, with an eye towards using a cache. Let the input consist of the four 16-bit integers y_3, y_2, y_1, y_0, with y_3 holding the most significant bits. Then the 16 least significant bits in the reverse come from y_3. To be precise, these bits appear in the reverse order in which they do in y_3. For example, if y_3 is (1110000000000001), then the 16 LSBs of the result are (1000000000000111).

Similar to computing parity (Problem 4.1 on Page 24), a very fast way to reverse bits for 16-bit integer when we are performing many reverses is to build an array-based lookup-table A such that for every 16-bit integer y, $A[y]$ holds the bit-reversal of y. We can then form the reverse of x with the reverse of y_0 in the most significant bit positions, followed by the reverse of y_1, followed by the reverse of y_2, followed by the reverse of y_3.

We illustrate the approach with 8-bit integers and 2-bit lookup table keys. The table is rev = $\langle(00), (10), (01), (11)\rangle$. If the input is (10010011), its reverse is rev(11), rev(00), rev(01), rev(10), i.e., (11001001).

```
def reverse_bits(x):
    MASK_SIZE = 16
    BIT_MASK = 0xFFFF
    return (PRECOMPUTED_REVERSE[x & BIT_MASK] << (3 * MASK_SIZE)
            | PRECOMPUTED_REVERSE[(x >> MASK_SIZE) & BIT_MASK] <<
            (2 * MASK_SIZE) |
            PRECOMPUTED_REVERSE[(x >> (2 * MASK_SIZE)) & BIT_MASK] << MASK_SIZE
            | PRECOMPUTED_REVERSE[(x >> (3 * MASK_SIZE)) & BIT_MASK])
```

The time complexity is identical to that for Solution 4.1 on Page 24, i.e., $O(n/L)$, for n-bit integers and L-bit cache keys.

4.4 Find a closest integer with the same weight

Define the *weight* of a nonnegative integer x to be the number of bits that are set to 1 in its binary representation. For example, since 92 in base-2 equals $(1011100)_2$, the weight of 92 is 4.

Write a program which takes as input a nonnegative integer x and returns a number y which is not equal to x, but has the same weight as x and their difference, $|y - x|$, is as small as possible. You can assume x is not 0, or all 1s. For example, if $x = 6$, you should return 5. You can assume the integer fits in 64 bits.

Hint: Start with the least significant bit.

Solution: A brute-force approach might be to try all integers $x-1, x+1, x-2, x+2, \ldots$, stopping as soon as we encounter one with the same weight at x. This performs very poorly on some inputs. One way to see this is to consider the case where $x = 2^3 = 8$. The only numbers with a weight of 1 are powers of 2. Thus, the algorithm will try the following sequence: $7, 9, 6, 10, 5, 11, 4$, stopping at 4 (since its weight is the same as 8's weight). The algorithm tries 2^{3-1} numbers smaller than 8, namely, $7, 6, 5, 4$, and $2^{3-1} - 1$ numbers greater than 8, namely, $9, 10, 11$. This example generalizes. Suppose $x = 2^{30}$. The power of 2 nearest to 2^{30} is 2^{29}. Therefore this computation will evaluate the weight of all integers between 2^{30} and 2^{29} and between 2^{30} and $2^{30} + 2^{29} - 1$, i.e., over one billion integers.

Heuristically, it is natural to focus on the LSB of the input, specifically, to swap the LSB with rightmost bit that differs from it. This yields the correct result for some inputs, e.g., for $(10)_2$ it returns $(01)_2$, which is the closest possible. However, more experimentation shows this heuristic does not work generally. For example, for $(111)_2$ (7 in decimal) it returns $(1110)_2$ which is 14 in decimal; however, $(1011)_2$ (11 in decimal) has the same weight, and is closer to $(111)_2$.

A little math leads to the correct approach. Suppose we flip the bit at index $k1$ and flip the bit at index $k2$, $k1 > k2$. Then the absolute value of the difference between the original integer and the new one is $2^{k1} - 2^{k2}$. To minimize this, we should make $k1$ as small as possible and $k2$ as close to $k1$.

Since we must preserve the weight, the bit at index $k1$ has to be different from the bit in $k2$, otherwise the flips lead to an integer with different weight. This means the smallest $k1$ is the rightmost bit that's different from the LSB, and $k2$ must be the very next bit. In summary, the correct approach is to swap the two rightmost consecutive bits that differ.

```
def closest_int_same_bit_count(x):
    NUM_UNSIGNED_BITS = 64
    for i in range(NUM_UNSIGNED_BITS - 1):
        if (x >> i) & 1 != (x >> (i + 1)) & 1:
            x ^= (1 << i) | (1 << (i + 1))  # Swaps bit-i and bit-(i + 1).
            return x

    # Raise error if all bits of x are 0 or 1.
    raise ValueError('All bits are 0 or 1')
```

The time complexity is $O(n)$, where n is the integer width.

Variant: Solve the same problem in $O(1)$ time and space.

4.5 Compute $x \times y$ without arithmetical operators

Sometimes the processors used in ultra low-power devices such as hearing aids do not have dedicated hardware for performing multiplication. A program that needs to perform multiplication must do so explicitly using lower-level primitives.

Write a program that multiplies two nonnegative integers. The only operators you are allowed to use are

- assignment,
- the bitwise operators »,«, |, &, ~, ^ and

- equality checks and Boolean combinations thereof.

You may use loops and functions that you write yourself. These constraints imply, for example, that you cannot use increment or decrement, or test if $x < y$.

Hint: Add using bitwise operations; multiply using shift-and-add.

Solution: A brute-force approach would be to perform repeated addition, i.e., initialize the result to 0 and then add x to it y times. For example, to form 5×3, we would start with 0 and repeatedly add 5, i.e., form $0+5, 5+5, 10+5$. The time complexity is very high—as much as $O(2^n)$, where n is the number of bits in the input, and it still leaves open the problem of adding numbers without the presence of an add instruction.

The algorithm taught in grade-school for decimal multiplication does not use repeated addition—it uses shift and add to achieve a much better time complexity. We can do the same with binary numbers—to multiply x and y we initialize the result to 0 and iterate through the bits of x, adding $2^k y$ to the result if the kth bit of x is 1.

The value $2^k y$ can be computed by left-shifting y by k. Since we cannot use add directly, we must implement it. We apply the grade-school algorithm for addition to the binary case, i.e., compute the sum bit-by-bit, and "rippling" the carry along.

As an example, we show how to multiply $13 = (1101)_2$ and $9 = (1001)_2$ using the algorithm described above. In the first iteration, since the LSB of 13 is 1, we set the result to $(1001)_2$. The second bit of $(1101)_2$ is 0, so we move on to the third bit. This bit is 1, so we shift $(1001)_2$ to the left by 2 to obtain $(100100)_2$, which we add to $(1001)_2$ to get $(101101)_2$. The fourth and final bit of $(1101)_2$ is 1, so we shift $(1001)_2$ to the left by 3 to obtain $(1001000)_2$, which we add to $(101101)_2$ to get $(1110101)_2 = 117$.

Each addition is itself performed bit-by-bit. For example, when adding $(101101)_2$ and $(1001000)_2$, the LSB of the result is 1 (since exactly one of the two LSBs of the operands is 1). The next bit is 0 (since both the next bits of the operands are 0). The next bit is 1 (since exactly one of the next bits of the operands is 1). The next bit is 0 (since both the next bits of the operands are 1). We also "carry" a 1 to the next position. The next bit is 1 (since the carry-in is 1 and both the next bits of the operands are 0). The remaining bits are assigned similarly.

```
def multiply(x, y):
    def add(a, b):
        running_sum, carryin, k, temp_a, temp_b = 0, 0, 1, a, b
        while temp_a or temp_b:
            ak, bk = a & k, b & k
            carryout = (ak & bk) | (ak & carryin) | (bk & carryin)
            running_sum |= ak ^ bk ^ carryin
            carryin, k, temp_a, temp_b = (carryout << 1, k << 1, temp_a >> 1,
                                          temp_b >> 1)
        return running_sum | carryin

    running_sum = 0
    while x:  # Examines each bit of x.
        if x & 1:
            running_sum = add(running_sum, y)
        x, y = x >> 1, y << 1
    return running_sum
```

The time complexity of addition is $O(n)$, where n is the number of bits needed to represent the operands. Since we do n additions to perform a single multiplication, the total time complexity is $O(n^2)$.

4.6 Compute x/y

Given two positive integers, compute their quotient, using only the addition, subtraction, and shifting operators.

Hint: Relate x/y to $(x - y)/y$.

Solution: A brute-force approach is to iteratively subtract y from x until what remains is less than y. The number of such subtractions is exactly the quotient, x/y, and the remainder is the term that's less than y. The complexity of the brute-force approach is very high, e.g., when $y = 1$ and $x = 2^{31} - 1$, it will take $2^{31} - 1$ iterations.

A better approach is to try and get more work done in each iteration. For example, we can compute the largest k such that $2^k y \leq x$, subtract $2^k y$ from x, and add 2^k to the quotient. For example, if $x = (1011)_2$ and $y = (10)_2$, then $k = 2$, since $2 \times 2^2 \leq 11$ and $2 \times 2^3 > 11$. We subtract $(1000)_2$ from $(1011)_2$ to get $(11)_2$, add $2^k = 2^2 = (100)_2$ to the quotient, and continue by updating x to $(11)_2$.

The advantage of using $2^k y$ is that it can be computed very efficiently using shifting, and x is at least halved in each iteration. If it takes n bits to represent x/y, there are $O(n)$ iterations. If the largest k such that $2^k y \leq x$ is computed by iterating through k, each iteration has time complexity $O(n)$. This leads to an $O(n^2)$ algorithm.

A better way to find the largest k in each iteration is to recognize that it keeps decreasing. Therefore, instead of testing in each iteration whether $2^0 y, 2^1 y, 2^2 y, \ldots$ is less than or equal to x, after we initially find the largest k such that $2^k y \leq x$, in subsequent iterations we test $2^{k-1} y, 2^{k-2} y, 2^{k-3} y, \ldots$ with x.

For the example given earlier, after setting the quotient to $(100)_2$ we continue with $(11)_2$. Now the largest k such that $2^k y \leq (11)_2$ is 0, so we add $2^0 = (1)_2$ to the quotient, which is now $(101)_2$. We continue with $(11)_2 - (10)_2 = (1)_2$. Since $(1)_2 < y$, we are done—the quotient is $(101)_2$ and the remainder is $(1)_2$.

```
def divide(x, y):
    result, power = 0, 32
    y_power = y << power
    while x >= y:
        while y_power > x:
            y_power >>= 1
            power -= 1

        result += 1 << power
        x -= y_power
    return result
```

In essence, the program applies the grade-school division algorithm to binary numbers. With each iteration, we process an additional bit. Therefore, assuming individual shift and add operations take $O(1)$ time, the time complexity is $O(n)$.

4.7 Compute x^y

Write a program that takes a double x and an integer y and returns x^y. You can ignore overflow and underflow.

Hint: Exploit mathematical properties of exponentiation.

Solution: First, assume y is nonnegative. The brute-force algorithm is to form $x^2 = x \times x$, then $x^3 = x^2 \times x$, and so on. This approach takes $y - 1$ multiplications, which is $O(2^n)$, where n is number of bits needed to represent y.

The key to efficiency is to try and get more work done with each multiplication, thereby using fewer multiplications to accomplish the same result. For example, to compute 1.1^{21}, instead of starting with 1.1 and multiplying by 1.1 20 times, we could multiply 1.1 by $1.1^2 = 1.21$ 10 times for a total of 11 multiplications (one to compute 1.1^2, and 10 additional multiplications by 1.21). We can do still better by computing 1.1^3, 1.1^4, etc.

When y is a power of 2, the approach that uses fewest multiplications is iterated squaring, i.e., forming $x, x^2, (x^2)^2 = x^4, (x^4)^2 = x^8, \ldots$. To develop an algorithm that works for general y, it is instructive to look at the binary representation of y, as well as properties of exponentiation, specifically $x^{y_0+y_1} = x^{y_0} \cdot x^{y_1}$.

We begin with some small concrete instances, first assuming that y is nonnegative. For example, $x^{(1010)_2} = x^{(101)_2+(101)_2} = x^{(101)_2} \times x^{(101)_2}$. Similarly, $x^{(101)_2} = x^{(100)_2+(1)_2} = x^{(100)_2} \times x = x^{(10)_2} \times x^{(10)_2} \times x$.

Generalizing, if the least significant bit of y is 0, the result is $(x^{y/2})^2$; otherwise, it is $x \times (x^{y/2})^2$. This gives us a recursive algorithm for computing x^y when y is nonnegative.

The only change when y is negative is replacing x by $1/x$ and y by $-y$. In the implementation below we replace the recursion with a while loop to avoid the overhead of function calls.

```
def power(x, y):
    result, power = 1.0, y
    if y < 0:
        power, x = -power, 1.0 / x
    while power:
        if power & 1:
            result *= x
        x, power = x * x, power >> 1
    return result
```

The number of multiplications is at most twice the index of y's MSB, implying an $O(n)$ time complexity.

4.8 Reverse digits

Write a program which takes an integer and returns the integer corresponding to the digits of the input written in reverse order. For example, the reverse of 42 is 24, and the reverse of -314 is -413.

Hint: How would you solve the same problem if the input is presented as a string?

Solution: The brute-force approach is to convert the input to a string, and then compute the reverse from the string by traversing it from back to front. For example, $(1100)_2$ is the decimal number 12, and the answer for $(1100)_2$ can be computed by traversing the string "12" in reverse order.

Closer analysis shows that we can avoid having to form a string. Consider the input 1132. The first digit of the result is 2, which we can obtain by taking the input mod 10. The remaining digits of the result are the reverse of 1132/10 = 113. Generalizing, let the input be k. If $k \geq 0$, then k mod 10 is the most significant digit of the result and the subsequent digits are the reverse of $\frac{k}{10}$. Continuing with the example, we iteratively update the result and the input as 2 and 113, then 23 and 11, then 231 and 1, then 2311.

For general k, we record its sign, solve the problem for $|k|$, and apply the sign to the result.

```
def reverse(x):
    result, x_remaining = 0, abs(x)
    while x_remaining:
        result = result * 10 + x_remaining % 10
        x_remaining //= 10
    return -result if x < 0 else result
```

The time complexity is $O(n)$, where n is the number of digits in k.

4.9 Check if a decimal integer is a palindrome

A palindromic string is one which reads the same forwards and backwards, e.g., "redivider". In this problem, you are to write a program which determines if the decimal representation of an integer is a palindromic string. For example, your program should return true for the inputs 0, 1, 7, 11, 121, 333, and 2147447412, and false for the inputs −1, 12, 100, and 2147483647.

Write a program that takes an integer and determines if that integer's representation as a decimal string is a palindrome.

Hint: It's easy to come up with a simple expression that extracts the least significant digit. Can you find a simple expression for the most significant digit?

Solution: First note that if the input is negative, then its representation as a decimal string cannot be palindromic, since it begins with a −.

A brute-force approach would be to convert the input to a string and then iterate through the string, pairwise comparing digits starting from the least significant digit and the most significant digit, and working inwards, stopping if there is a mismatch. The time and space complexity are $O(n)$, where n is the number of digits in the input.

We can avoid the $O(n)$ space complexity used by the string representation by directly extracting the digits from the input. The number of digits, n, in the input's string representation is the log (base 10) of the input value, x. To be precise, $n = \lfloor \log_{10} x \rfloor + 1$. Therefore, the least significant digit is x mod 10, and the most significant digit is $x/10^{n-1}$. In the program below, we iteratively compare the most and least significant digits, and then remove them from the input. For example, if the input is 151751, we would compare the leading and trailing digits, 1 and 1. Since these are equal, we update the value to 5175. The leading and trailing digits are equal, so we update to 17. Now the leading and trailing are unequal, so we return false. If instead the number was 157751, the final compare would be of 7 with 7, so we would return true.

```
def is_palindrome_number(x):
    if x <= 0:
```

```
    return x == 0

num_digits = math.floor(math.log10(x)) + 1
msd_mask = 10**(num_digits - 1)
for i in range(num_digits // 2):
    if x // msd_mask != x % 10:
        return False
    x %= msd_mask  # Remove the most significant digit of x.
    x //= 10  # Remove the least significant digit of x.
    msd_mask //= 100
return True
```

The time complexity is $O(n)$, and the space complexity is $O(1)$. Alternatively, we could use Solution 4.8 on Page 32 to reverse the digits in the number and see if it is unchanged.

4.10 Generate uniform random numbers

This problem is motivated by the following scenario. Six friends have to select a designated driver using a single unbiased coin. The process should be fair to everyone.

How would you implement a random number generator that generates a random integer i between a and b, inclusive, given a random number generator that produces zero or one with equal probability? All values in $[a, b]$ should be equally likely.

Hint: How would you mimic a three-sided coin with a two-sided coin?

Solution: Note that it is easy to produce a random integer between 0 and 2^i-1, inclusive: concatenate i bits produced by the random number generator. For example, two calls to the random number generator will produce one of $(00)_2, (01)_2, (10)_2, (11)_2$. These four possible outcomes encode the four integers $0, 1, 2, 3$, and all of them are equally likely.

For the general case, first note that it is equivalent to produce a random integer between 0 and $b-a$, inclusive, since we can simply add a to the result. If $b-a$ is equal to 2^i-1, for some i, then we can use the approach in the previous paragraph.

If $b-a$ is not of the form 2^i-1, we find the smallest number of the form 2^i-1 that is greater than $b-a$. We generate an i-bit number as before. This i-bit number may or may not lie between 0 and $b-a$, inclusive. If it is within the range, we return it—all such numbers are equally likely. If it is not within the range, we try again with i new random bits. We keep trying until we get a number within the range.

For example, to generate a random number corresponding to a dice roll, i.e., a number between 1 and 6, we begin by making three calls to the random number generator (since $2^2-1 < (6-1) \leq 2^3-1$). If this yields one of $(000)_2, (001)_2, (010)_2, (011)_2, (100)_2, (101)_2$, we return 1 plus the corresponding value. Observe that all six values between 1 and 6, inclusive, are equally likely to be returned. If the three calls yields one of $(110)_2, (111)_2$, we make three more calls. Note that the probability of having to try again is 2/8, which is less than half. Since successive calls are independent, the probability that we require many attempts diminishes very rapidly, e.g., the probability of not getting a result in 10 attempts is $(2/8)^{10}$ which is less than one-in-a-million.

```
def uniform_random(lower_bound, upper_bound):
```

```
number_of_outcomes = upper_bound - lower_bound + 1
while True:
    result, i = 0, 0
    while (1 << i) < number_of_outcomes:
        # zero_one_random() is the provided random number generator.
        result = (result << 1) | zero_one_random()
        i += 1
    if result < number_of_outcomes:
        break
return result + lower_bound
```

To analyze the time complexity, let $t = b - a + 1$. The probability that we succeed in the first try is $t/2^i$. Since 2^i is the smallest power of 2 greater than or equal to t, it must be less than $2t$. (An easy way to see this is to consider the binary representation of t and $2t$.) This implies that $t/2^i > t/2t = (1/2)$. Hence the probability that we do not succeed on the first try is $1 - t/2^i < 1/2$. Since successive tries are independent, the probability that more than k tries are needed is less than or equal to $1/2^k$. Hence, the expected number of tries is not more than $1 + 2(1/2)^1 + 3(1/2)^2 + \ldots$. The series converges, so the number of tries is $O(1)$. Each try makes $\lceil \log(b - a + 1) \rceil$ calls to the 0/1-valued random number generator. Assuming the 0/1-valued random number generator takes $O(1)$ time, the time complexity is $O(\log(b - a + 1))$.

4.11 Rectangle intersection

This problem is concerned with rectangles whose sides are parallel to the X-axis and Y-axis. See Figure 4.2 for examples.

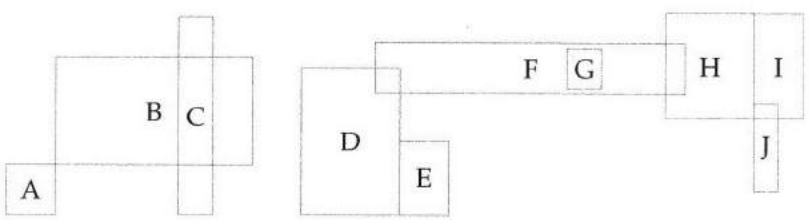


Figure 4.2: Examples of XY-aligned rectangles.

Write a program which tests if two rectangles have a nonempty intersection. If the intersection is nonempty, return the rectangle formed by their intersection.

Hint: Think of the X and Y dimensions independently.

Solution: Since the problem leaves it unspecified, we will treat the boundary as part of the rectangle. This implies, for example, rectangles A and B in Figure 4.2 intersect.

There are many qualitatively different ways in which rectangles can intersect, e.g., they have partial overlap (D and F), one contains the other (F and G), they share a common side (D and E), they share a common corner (A and B), they form a cross (B and C), they form a tee (F and H), etc. The case analysis is quite tricky.

A better approach is to focus on conditions under which it can be guaranteed that the rectangles do *not* intersect. For example, the rectangle with left-most lower point $(1, 2)$, width 3, and height 4

cannot possibly intersect with the rectangle with left-most lower point $(5, 3)$, width 2, and height 4, since the X-values of the first rectangle range from 1 to $1 + 3 = 4$, inclusive, and the X-values of the second rectangle range from 5 to $5 + 2 = 7$, inclusive.

Similarly, if the Y-values of the first rectangle do not intersect with the Y-values of the second rectangle, the two rectangles cannot intersect.

Equivalently, if the set of X-values for the rectangles intersect and the set of Y-values for the rectangles intersect, then all points with those X- and Y-values are common to the two rectangles, so there is a nonempty intersection.

```
Rectangle = collections.namedtuple('Rectangle', ('x', 'y', 'width', 'height'))


def intersect_rectangle(R1, R2):
    def is_intersect(R1, R2):
        return (R1.x <= R2.x + R2.width and R1.x + R1.width >= R2.x
                and R1.y <= R2.y + R2.height and R1.y + R1.height >= R2.y)

    if not is_intersect(R1, R2):
        return Rectangle(0, 0, -1, -1)  # No intersection.
    return Rectangle(
        max(R1.x, R2.x),
        max(R1.y, R2.y),
        min(R1.x + R1.width, R2.x + R2.width) - max(R1.x, R2.x),
        min(R1.y + R1.height, R2.y + R2.height) - max(R1.y, R2.y))
```

The time complexity is $O(1)$, since the number of operations is constant.

Variant: Given four points in the plane, how would you check if they are the vertices of a rectangle?

Variant: How would you check if two rectangles, not necessarily aligned with the X and Y axes, intersect?

CHAPTER

5 Arrays

The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine.

— *"Intelligent Machinery,"*
A. M. TURING, 1948

The simplest data structure is the array, which is a contiguous block of memory. It is usually used to represent sequences. Given an array A, $A[i]$ denotes the $(i+1)$th object stored in the array. Retrieving and updating $A[i]$ takes $O(1)$ time. Insertion into a full array can be handled by resizing, i.e., allocating a new array with additional memory and copying over the entries from the original array. This increases the worst-case time of insertion, but if the new array has, for example, a constant factor larger than the original array, the average time for insertion is constant since resizing is infrequent. Deleting an element from an array entails moving all successive elements one over to the left to fill the vacated space. For example, if the array is $\langle 2,3,5,7,9,11,13,17\rangle$, then deleting the element at index 4 results in the array $\langle 2,3,5,7,11,13,17,0\rangle$. (We do not care about the last value.) The time complexity to delete the element at index i from an array of length n is $O(n-i)$. The same is true for inserting a new element (as opposed to updating an existing entry).

Array boot camp

The following problem gives good insight into working with arrays: Your input is an array of integers, and you have to reorder its entries so that the even entries appear first. This is easy if you use $O(n)$ space, where n is the length of the array. However, you are required to solve it without allocating additional storage.

When working with arrays you should take advantage of the fact that you can operate efficiently on both ends. For this problem, we can partition the array into three subarrays: Even, Unclassified, and Odd, appearing in that order. Initially Even and Odd are empty, and Unclassified is the entire array. We iterate through Unclassified, moving its elements to the boundaries of the Even and Odd subarrays via swaps, thereby expanding Even and Odd, and shrinking Unclassified.

```
def even_odd(A):
    next_even, next_odd = 0, len(A) - 1
    while next_even < next_odd:
        if A[next_even] % 2 == 0:
            next_even += 1
        else:
            A[next_even], A[next_odd] = A[next_odd], A[next_even]
            next_odd -= 1
```

The additional space complexity is clearly $O(1)$—a couple of variables that hold indices, and a temporary variable for performing the swap. We do a constant amount of processing per entry, so the time complexity is $O(n)$.

Array problems often have simple brute-force solutions that use $O(n)$ space, but there are subtler solutions that **use the array itself** to **reduce space** complexity to $O(1)$.
Filling an array from the front is slow, so see if it's possible to **write values from the back**.
Instead of deleting an entry (which requires moving all entries to its left), consider **overwriting** it.
When dealing with integers encoded by an array consider **processing the digits from the back** of the array. Alternately, reverse the array so the **least-significant digit is the first entry**.
Be comfortable with writing code that operates on **subarrays**.
It's incredibly easy to make **off-by-1** errors when operating on arrays—reading past the last element of an array is a common error which has catastrophic consequences.
Don't worry about preserving the **integrity** of the array (sortedness, keeping equal entries together, etc.) until it is time to return.
An array can serve as a good data structure when you know the distribution of the elements in advance. For example, a Boolean array of length W is a good choice for representing a **subset of** $\{0, 1, \ldots, W-1\}$. (When using a Boolean array to represent a subset of $\{1, 2, 3, \ldots, n\}$, allocate an array of size $n+1$ to simplify indexing.) .
When operating on $2D$ arrays, **use parallel logic** for rows and for columns.
Sometimes it's easier to **simulate the specification**, than to analytically solve for the result. For example, rather than writing a formula for the i-th entry in the spiral order for an $n \times n$ matrix, just compute the output from the beginning.

Table 5.1: Top Tips for Arrays

Know your array libraries

Arrays in Python are provided by the `list` type. (The `tuple` type is very similar to the `list` type, with the constraint that it is immutable.) The key property of a `list` is that it is dynamically-resized, i.e., there's no bound as to how many values can be added to it. In the same way, values can be deleted and inserted at arbitrary locations.

- Know the syntax for instantiating a `list`, e.g., `[3, 5, 7, 11]`, `[1] + [0] * 10`, `list(range(100))`. (List comprehension, described later, is also a powerful tool for instantiating arrays.)
- The basic operations are `len(A)`, `A.append(42)`, `A.remove(2)`, and `A.insert(3, 28)`.
- Know how to instantiate a $2D$ array, e.g., `[[1, 2, 4], [3, 5, 7, 9], [13]]`.

- Checking if a value is present in an array is as simple as `a in A`. (This operation has $O(n)$ time complexity, where n is the length of the array.)
- Understand how copy works, i.e., the difference between `B = A` and `B = list(A)`. Understand what a deep copy is, and how it differs from a shallow copy, i.e., how `copy.copy(A)` differs from `copy.deepcopy(A)`.
- Key methods for `list` include `min(A)`, `max(A)`, binary search for sorted `lists` (`bisect.bisect(A, 6)`, `bisect.bisect_left(A, 6)`, and `bisect.bisect_right(A, 6)`), `A.reverse()` (in-place), `reversed(A)` (returns an iterator), `A.sort()` (in-place), `sorted(A)` (returns a copy), `del A[i]` (deletes the i-th element), and `del A[i:j]` (removes the slice).
- Slicing is a very succinct way of manipulating arrays. It can be viewed as a generalization of indexing: the most general form of slice is `A[i:j:k]`, with all of `i`, `j`, and `k` being optional. Let `A = [1, 6, 3, 4, 5, 2, 7]`. Here are some examples of slicing: `A[2:4]` is `[3, 4]`, `A[2:]` is `[3, 4, 5, 2, 7]`, `A[:4]` is `[1, 6, 3, 4]`, `A[:-1]` is `[1, 6, 3, 4, 5, 2]`, `A[-3:]` is `[5, 2, 7]`, `A[-3:-1]` is `[5, 2]`, `A[1:5:2]` is `[6, 4]`, `A[5:1:-2]` is `[2, 4]`, and `A[::-1]` is `[7, 2, 5, 4, 3, 6, 1]` (reverses list). Slicing can also be used to rotate a list: `A[k:] + A[:k]` rotates `A` by k to the left. It can also be used to create a copy: `B = A[:]` does a (shallow) copy of `A` into `B`.
- Python provides a feature called list comprehension that is a succinct way to create lists. A list comprehension consists of (1.) an input sequence, (2.) an iterator over the input sequence, (3.) a logical condition over the iterator (this is optional), and (4.) an expression that yields the elements of the derived list. For example, `[x**2 for x in range(6)]` yields `[0, 1, 4, 9, 16, 25]`, and `[x**2 for x in range(6) if x % 2 == 0]` yields `[0, 4, 16]`.

  Although list comprehensions can always be rewritten using `map()`, `filter()`, and lambdas, they are clearer to read, in large part because they do not need lambdas.

  List comprehension supports multiple levels of looping. This can be used to create the product of sets, e.g., if `A = [1, 3, 5]` and `B = ['a', 'b']`, then `[(x, y) for x in A for y in B]` creates `[(1, 'a'), (1, 'b'), (3, 'a'), (3, 'b'), (5, 'a'), (5, 'b')]`. It can also be used to convert a $2D$ `list` to a $1D$ `list`, e.g., if `M = [['a', 'b', 'c'], ['d', 'e', 'f']]`, `x for row in M for x in row` creates `['a', 'b', 'c', 'd', 'e', 'f']`. Two levels of looping also allow for iterating over each entry in a $2D$ `list`, e.g., if `A = [[1, 2, 3], [4, 5, 6]]` then `[[x**2 for x in row] for row in M]` yields `[[1, 4, 9], [16, 25, 36]]`.

  As a general rule, it is best to avoid more than two nested comprehensions, and use conventional nested for loops—the indentation makes it easier to read the program.

  Finally, sets and dictionaries also support list comprehensions, with the same benefits.

5.1 The Dutch national flag problem

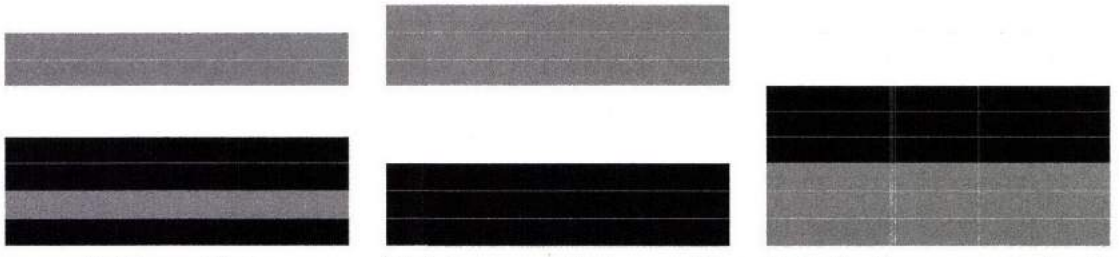
The quicksort algorithm for sorting arrays proceeds recursively—it selects an element (the "pivot"), reorders the array to make all the elements less than or equal to the pivot appear first, followed by all the elements greater than the pivot. The two subarrays are then sorted recursively.

Implemented naively, quicksort has large run times and deep function call stacks on arrays with many duplicates because the subarrays may differ greatly in size. One solution is to reorder the array so that all elements less than the pivot appear first, followed by elements equal to the pivot,

followed by elements greater than the pivot. This is known as Dutch national flag partitioning, because the Dutch national flag consists of three horizontal bands, each in a different color.

As an example, assuming that black precedes white and white precedes gray, Figure 5.1(b) is a valid partitioning for Figure 5.1(a). If gray precedes black and black precedes white, Figure 5.1(c) is a valid partitioning for Figure 5.1(a).

Generalizing, suppose $A = \langle 0, 1, 2, 0, 2, 1, 1 \rangle$, and the pivot index is 3. Then $A[3] = 0$, so $\langle 0, 0, 1, 2, 2, 1, 1 \rangle$ is a valid partitioning. For the same array, if the pivot index is 2, then $A[2] = 2$, so the arrays $\langle 0, 1, 0, 1, 1, 2, 2 \rangle$ as well as $\langle 0, 0, 1, 1, 1, 2, 2 \rangle$ are valid partitionings.



(a) Before partitioning.

(b) A three-way partitioning resembling the Dutch national flag.

(c) Another three-way partitioning: the Russian national flag.

Figure 5.1: Illustrating the Dutch national flag problem.

Write a program that takes an array A and an index i into A, and rearranges the elements such that all elements less than $A[i]$ (the "pivot") appear first, followed by elements equal to the pivot, followed by elements greater than the pivot.

Hint: Think about the partition step in quicksort.

Solution: The problem is trivial to solve with $O(n)$ additional space, where n is the length of A. We form three lists, namely, elements less than the pivot, elements equal to the pivot, and elements greater than the pivot. Consequently, we write these values into A. The time complexity is $O(n)$.

We can avoid using $O(n)$ additional space at the cost of increased time complexity as follows. In the first stage, we iterate through A starting from index 0, then index 1, etc. In each iteration, we seek an element smaller than the pivot—as soon as we find it, we move it to the subarray of smaller elements via an exchange. This moves all the elements less than the pivot to the start of the array. The second stage is similar to the first one, the difference being that we move elements greater than the pivot to the end of the array. Code illustrating this approach is shown below.

```
RED, WHITE, BLUE = range(3)


def dutch_flag_partition(pivot_index, A):
    pivot = A[pivot_index]
    # First pass: group elements smaller than pivot.
    for i in range(len(A)):
        # Look for a smaller element.
        for j in range(i + 1, len(A)):
            if A[j] < pivot:
                A[i], A[j] = A[j], A[i]
                break
```

```
    # Second pass: group elements larger than pivot.
    for i in reversed(range(len(A))):
        if A[i] < pivot:
            break
        # Look for a larger element. Stop when we reach an element less than
        # pivot, since first pass has moved them to the start of A.
        for j in reversed(range(i)):
            if A[j] > pivot:
                A[i], A[j] = A[j], A[i]
                break
```

The additional space complexity is now $O(1)$, but the time complexity is $O(n^2)$, e.g., if $i = n/2$ and all elements before i are greater than $A[i]$, and all elements after i are less than $A[i]$. Intuitively, this approach has bad time complexity because in the first pass when searching for each additional element smaller than the pivot we start from the beginning. However, there is no reason to start from so far back—we can begin from the last location we advanced to. (Similar comments hold for the second pass.)

To improve time complexity, we make a single pass and move all the elements less than the pivot to the beginning. In the second pass we move the larger elements to the end. It is easy to perform each pass in a single iteration, moving out-of-place elements as soon as they are discovered.

```
RED, WHITE, BLUE = range(3)


def dutch_flag_partition(pivot_index, A):
    pivot = A[pivot_index]
    # First pass: group elements smaller than pivot.
    smaller = 0
    for i in range(len(A)):
        if A[i] < pivot:
            A[i], A[smaller] = A[smaller], A[i]
            smaller += 1
    # Second pass: group elements larger than pivot.
    larger = len(A) - 1
    for i in reversed(range(len(A))):
        if A[i] < pivot:
            break
        elif A[i] > pivot:
            A[i], A[larger] = A[larger], A[i]
            larger -= 1
```

The time complexity is $O(n)$ and the space complexity is $O(1)$.

The algorithm we now present is similar to the one sketched above. The main difference is that it performs classification into elements less than, equal to, and greater than the pivot in a single pass. This reduces runtime, at the cost of a trickier implementation. We do this by maintaining four subarrays: *bottom* (elements less than pivot), *middle* (elements equal to pivot), *unclassified*, and *top* (elements greater than pivot). Initially, all elements are in *unclassified*. We iterate through elements in *unclassified*, and move elements into one of *bottom*, *middle*, and *top* groups according to the relative order between the incoming unclassified element and the pivot.

As a concrete example, suppose the array is currently $A = \langle -3, 0, -1, 1, 1, ?, ?, ?, 4, 2 \rangle$, where the pivot is 1 and ? denotes unclassified elements. There are three possibilities for the first unclassified element, $A[5]$.

- $A[5]$ is less than the pivot, e.g., $A[5] = -5$. We exchange it with the first 1, i.e., the new array is $\langle -3, 0, -1, -5, 1, 1, ?, ?, 4, 2 \rangle$.
- $A[5]$ is equal to the pivot, i.e., $A[5] = 1$. We do not need to move it, we just advance to the next unclassified element, i.e., the array is $\langle -3, 0, -1, 1, 1, 1, ?, ?, 4, 2 \rangle$.
- $A[5]$ is greater than the pivot, e.g., $A[5] = 3$. We exchange it with the last unclassified element, i.e., the new array is $\langle -3, 0, -1, 1, 1, ?, ?, 3, 4, 2 \rangle$.

Note how the number of unclassified elements reduces by one in each case.

```
RED, WHITE, BLUE = range(3)


def dutch_flag_partition(pivot_index, A):
    pivot = A[pivot_index]
    # Keep the following invariants during partitioning:
    # bottom group: A[:smaller].
    # middle group: A[smaller:equal].
    # unclassified group: A[equal:larger].
    # top group: A[larger:].
    smaller, equal, larger = 0, 0, len(A)
    # Keep iterating as long as there is an unclassified element.,
    while equal < larger:
        # A[equal] is the incoming unclassified element.
        if A[equal] < pivot:
            A[smaller], A[equal] = A[equal], A[smaller]
            smaller, equal = smaller + 1, equal + 1
        elif A[equal] == pivot:
            equal += 1
        else:  # A[equal] > pivot.
            larger -= 1
            A[equal], A[larger] = A[larger], A[equal]
```

Each iteration decreases the size of *unclassified* by 1, and the time spent within each iteration is $O(1)$, implying the time complexity is $O(n)$. The space complexity is clearly $O(1)$.

Variant: Assuming that keys take one of three values, reorder the array so that all objects with the same key appear together. The order of the subarrays is not important. For example, both Figures 5.1(b) and 5.1(c) on Page 40 are valid answers for Figure 5.1(a) on Page 40. Use $O(1)$ additional space and $O(n)$ time.

Variant: Given an array A of n objects with keys that takes one of four values, reorder the array so that all objects that have the same key appear together. Use $O(1)$ additional space and $O(n)$ time.

Variant: Given an array A of n objects with Boolean-valued keys, reorder the array so that objects that have the key false appear first. Use $O(1)$ additional space and $O(n)$ time.

Variant: Given an array A of n objects with Boolean-valued keys, reorder the array so that objects that have the key false appear first. The relative ordering of objects with key true should not change. Use $O(1)$ additional space and $O(n)$ time.

5.2 Increment an arbitrary-precision integer

Write a program which takes as input an array of digits encoding a nonnegative decimal integer D and updates the array to represent the integer $D + 1$. For example, if the input is $\langle 1, 2, 9 \rangle$ then you should update the array to $\langle 1, 3, 0 \rangle$. Your algorithm should work even if it is implemented in a language that has finite-precision arithmetic.

Hint: Experiment with concrete examples.

Solution: A brute-force approach might be to convert the array of digits to the equivalent integer, increment that, and then convert the resulting value back to an array of digits. For example, if the array is $\langle 1, 2, 9 \rangle$, we would derive the integer 129, add one to get 130, then extract its digits to form $\langle 1, 3, 0 \rangle$. When implemented in a language that imposes a limit on the range of values an integer type can take, this approach will fail on inputs that encode integers outside of that range.

We can avoid overflow issues by operating directly on the array of digits. Specifically, we mimic the grade-school algorithm for adding integers, which entails adding digits starting from the least significant digit, and propagate carries. If the result has an additional digit, e.g., $99 + 1 = 100$, there is not enough storage in the array for the result—we need three digits to represent 100, but the input has only two digits.

For the given example, we would update 9 to 0 with a carry-out of 1. We update 2 to 3 (because of the carry-in). There is no carry-out, so we stop—the result is $\langle 1, 3, 0 \rangle$.

```
def plus_one(A):
    A[-1] += 1
    for i in reversed(range(1, len(A))):
        if A[i] != 10:
            break
        A[i] = 0
        A[i - 1] += 1
    if A[0] == 10:
        # There is a carry-out, so we need one more digit to store the result.
        # A slick way to do this is to append a 0 at the end of the array,
        # and update the first entry to 1.
        A[0] = 1
        A.append(0)
    return A
```

The time complexity is $O(n)$, where n is the length of A.

Variant: Write a program which takes as input two strings s and t of bits encoding binary numbers B_s and B_t, respectively, and returns a new string of bits representing the number $B_s + B_t$.

5.3 Multiply two arbitrary-precision integers

Certain applications require arbitrary precision arithmetic. One way to achieve this is to use arrays to represent integers, e.g., with one digit per array entry, with the most significant digit appearing first, and a negative leading digit denoting a negative integer. For example, $\langle 1, 9, 3, 7, 0, 7, 7, 2, 1 \rangle$ represents 193707721 and $\langle -7, 6, 1, 8, 3, 8, 2, 5, 7, 2, 8, 7 \rangle$ represents -761838257287.

Write a program that takes two arrays representing integers, and returns an integer representing their product. For example, since $193707721 \times -761838257287 = -147573952589676412927$, if the inputs are $\langle 1,9,3,7,0,7,7,2,1 \rangle$ and $\langle -7,6,1,8,3,8,2,5,7,2,8,7 \rangle$, your function should return $\langle -1,4,7,5,7,3,9,5,2,5,8,9,6,7,6,4,1,2,9,2,7 \rangle$.

Hint: Use arrays to simulate the grade-school multiplication algorithm.

Solution: As in Solution 5.2 on the preceding page, the possibility of overflow precludes us from converting to the integer type.

Instead we can use the grade-school algorithm for multiplication which consists of multiplying the first number by each digit of the second, and then adding all the resulting terms.

From a space perspective, it is better to incrementally add the terms rather than compute all of them individually and then add them up. The number of digits required for the product is at most $n + m$ for n and m digit operands, so we use an array of size $n + m$ for the result.

For example, when multiplying 123 with 987, we would form $7 \times 123 = 861$, then we would form $8 \times 123 \times 10 = 9840$, which we would add to 861 to get 10701. Then we would form $9 \times 123 \times 100 = 110700$, which we would add to 10701 to get the final result 121401. (All numbers shown are represented using arrays of digits.)

```
def multiply(num1, num2):
    sign = -1 if (num1[0] < 0) ^ (num2[0] < 0) else 1
    num1[0], num2[0] = abs(num1[0]), abs(num2[0])

    result = [0] * (len(num1) + len(num2))
    for i in reversed(range(len(num1))):
        for j in reversed(range(len(num2))):
            result[i + j + 1] += num1[i] * num2[j]
            result[i + j] += result[i + j + 1] // 10
            result[i + j + 1] %= 10

    # Remove the leading zeroes.
    result = result[next((i for i, x in enumerate(result)
                          if x != 0), len(result)):] or [0]
    return [sign * result[0]] + result[1:]
```

There are m partial products, each with at most $n + 1$ digits. We perform $O(1)$ operations on each digit in each partial product, so the time complexity is $O(nm)$.

5.4 Advancing through an array

In a particular board game, a player has to try to advance through a sequence of positions. Each position has a nonnegative integer associated with it, representing the maximum you can advance from that position in one move. You begin at the first position, and win by getting to the last position. For example, let $A = \langle 3,3,1,0,2,0,1 \rangle$ represent the board game, i.e., the ith entry in A is the maximum we can advance from i. Then the game can be won by the following sequence of advances through A: take 1 step from $A[0]$ to $A[1]$, then 3 steps from $A[1]$ to $A[4]$, then 2 steps from $A[4]$ to $A[6]$, which is the last position. Note that $A[0] = 3 \geq 1$, $A[1] = 3 \geq 3$, and $A[4] = 2 \geq 2$, so all moves are valid. If A instead was $\langle 3,2,0,0,2,0,1 \rangle$, it would not possible to advance past position 3, so the game cannot be won.

Write a program which takes an array of n integers, where $A[i]$ denotes the maximum you can advance from index i, and returns whether it is possible to advance to the last index starting from the beginning of the array.

Hint: Analyze each location, starting from the beginning.

Solution: It is natural to try advancing as far as possible in each step. This approach does not always work, because it potentially skips indices containing large entries. For example, if $A = \langle 2,4,1,1,0,2,3 \rangle$, then it advances to index 2, which contains a 1, which leads to index 3, after which it cannot progress. However, advancing to index 1, which contains a 4 lets us proceed to index 5, from which we can advance to index 6.

The above example suggests iterating through all entries in A. As we iterate through the array, we track the furthest index we know we can advance to. The furthest we can advance from index i is $i + A[i]$. If, for some i before the end of the array, i is the furthest index that we have demonstrated that we can advance to, we cannot reach the last index. Otherwise, we reach the end.

For example, if $A = \langle 3,3,1,0,2,0,1 \rangle$, we iteratively compute the furthest we can advance to as $0,3,4,4,4,6,6,7$, which reaches the last index, 6. If $A = \langle 3,2,0,0,2,0,1 \rangle$, we iteratively update the furthest we can advance to as $0,3,3,3,3$, after which we cannot advance, so it is not possible to reach the last index.

The code below implements this algorithm. Note that it is robust with respect to negative entries, since we track the maximum of how far we proved we can advance to and $i + A[i]$.

```
def can_reach_end(A):
    furthest_reach_so_far, last_index = 0, len(A) - 1
    i = 0
    while i <= furthest_reach_so_far and furthest_reach_so_far < last_index:
        furthest_reach_so_far = max(furthest_reach_so_far, A[i] + i)
        i += 1
    return furthest_reach_so_far >= last_index
```

The time complexity is $O(n)$, and the additional space complexity (beyond what is used for A) is three integer variables, i.e., $O(1)$.

Variant: Write a program to compute the minimum number of steps needed to advance to the last location.

5.5 Delete duplicates from a sorted array

This problem is concerned with deleting repeated elements from a sorted array. For example, for the array $\langle 2,3,5,5,7,11,11,11,13 \rangle$, then after deletion, the array is $\langle 2,3,5,7,11,13,0,0,0 \rangle$. After deleting repeated elements, there are 6 valid entries. There are no requirements as to the values stored beyond the last valid element.

Write a program which takes as input a sorted array and updates it so that all duplicates have been removed and the remaining elements have been shifted left to fill the emptied indices. Return the number of valid elements. Many languages have library functions for performing this operation—you cannot use these functions.

Hint: There is an $O(n)$ time and $O(1)$ space solution.

Solution: Let A be the array and n its length. If we allow ourselves $O(n)$ additional space, we can solve the problem by iterating through A and recording values that have not appeared previously into a hash table. (The hash table is used to determine if a value is new.) New values are also written to a list. The list is then copied back into A.

Here is a brute-force algorithm that uses $O(1)$ additional space—iterate through A, testing if $A[i]$ equals $A[i+1]$, and, if so, shift all elements at and after $i+2$ to the left by one. When all entries are equal, the number of shifts is $(n-1)+(n-2)+\cdots+2+1$, i.e., $O(n^2)$, where n is the length of the array.

The intuition behind achieving a better time complexity is to reduce the amount of shifting. Since the array is sorted, repeated elements must appear one-after-another, so we do not need an auxiliary data structure to check if an element has appeared already. We move just one element, rather than an entire subarray, and ensure that we move it just once.

For the given example, $\langle 2,3,5,5,7,11,11,11,13\rangle$, when processing the $A[3]$, since we already have a 5 (which we know by comparing $A[3]$ with $A[2]$), we advance to $A[4]$. Since this is a new value, we move it to the first vacant entry, namely $A[3]$. Now the array is $\langle 2,3,5,7,7,11,11,11,13\rangle$, and the first vacant entry is $A[4]$. We continue from $A[5]$.

```
# Returns the number of valid entries after deletion.
def delete_duplicates(A):
    if not A:
        return 0

    write_index = 1
    for i in range(1, len(A)):
        if A[write_index - 1] != A[i]:
            A[write_index] = A[i]
            write_index += 1
    return write_index
```

The time complexity is $O(n)$, and the space complexity is $O(1)$, since all that is needed is the two additional variables.

Variant: Implement a function which takes as input an array and a key, and updates the array so that all occurrences of the input key have been removed and the remaining elements have been shifted left to fill the emptied indices. Return the number of remaining elements. There are no requirements as to the values stored beyond the last valid element.

Variant: Write a program which takes as input a sorted array A of integers and a positive integer m, and updates A so that if x appears m times in A it appears exactly $\min(2,m)$ times in A. The update to A should be performed in one pass, and no additional storage may be allocated.

5.6 Buy and sell a stock once

This problem is concerned with the problem of optimally buying and selling a stock once, as described on Page 2. As an example, consider the following sequence of stock prices: $\langle 310,315,275,295,260,270,290,230,255,250\rangle$. The maximum profit that can be made with one buy and one sell is 30—buy at 260 and sell at 290. Note that 260 is not the lowest price, nor 290 the highest price.

Write a program that takes an array denoting the daily stock price, and returns the maximum profit that could be made by buying and then selling one share of that stock. There is no need to buy if no profit is possible.

Hint: Identifying the minimum and maximum is not enough since the minimum may appear after the maximum height. Focus on valid differences.

Solution: We developed several algorithms for this problem in the introduction. Specifically, on Page 2 we showed how to compute the maximum profit by computing the difference of the current entry with the minimum value seen so far as we iterate through the array.

For example, the array of minimum values seen so far for the given example is $\langle 310, 310, 275, 275, 260, 260, 260, 230, 230, 230 \rangle$. The maximum profit that can be made by selling on each specific day is the difference of the current price and the minimum seen so far, i.e., $\langle 0, 5, 0, 20, 0, 10, 30, 0, 25, 20 \rangle$. The maximum profit overall is 30, corresponding to buying 260 and selling for 290.

```
def buy_and_sell_stock_once(prices):
    min_price_so_far, max_profit = float('inf'), 0.0
    for price in prices:
        max_profit_sell_today = price - min_price_so_far
        max_profit = max(max_profit, max_profit_sell_today)
        min_price_so_far = min(min_price_so_far, price)
    return max_profit
```

The time complexity is $O(n)$ and the space complexity is $O(1)$, where n is the length of the array.

Variant: Write a program that takes an array of integers and finds the length of a longest subarray all of whose entries are equal.

5.7 Buy and sell a stock twice

The max difference problem, introduced on Page 1, formalizes the maximum profit that can be made by buying and then selling a single share over a given day range.

Write a program that computes the maximum profit that can be made by buying and selling a share at most twice. The second buy must be made on another date after the first sale.

Hint: What do you need to know about the first i elements when processing the $(i+1)$th element?

Solution: The brute-force algorithm which examines all possible combinations of buy-sell-buy-sell days has complexity $O(n^4)$. The complexity can be improved to $O(n^2)$ by applying the $O(n)$ algorithm to each pair of subarrays formed by splitting A.

The inefficiency in the above approaches comes from not taking advantage of previous computations. Suppose we record the best solution for $A[0, j]$, j between 1 and $n-1$, inclusive. Now we can do a reverse iteration, computing the best solution for a single buy-and-sell for $A[j, n-1]$, j between 1 and $n-1$, inclusive. For each day, we combine this result with the result from the forward iteration for the previous day—this yields the maximum profit if we buy and sell once before the current day and once at or after the current day.

For example, suppose the input array is $\langle 12, 11, 13, 9, 12, 8, 14, 13, 15 \rangle$. Then the most profit that can be made with a single buy and sell by Day i (inclusive) is $F = \langle 0, 0, 2, 2, 3, 3, 6, 6, 7 \rangle$. Working

backwards, the most profit that can be made with a single buy and sell on or after Day i is $B = \langle 7,7,7,7,7,7,2,2,0 \rangle$. To combine these two, we compute $M[i] = F[i-1] + B[i]$, where $F[-1]$ is taken to be 0 (since the second buy must happen strictly after the first sell). This yields $M = \langle 7,7,7,9,9,10,5,8,6 \rangle$, i.e., the maximum profit is 10.

```
def buy_and_sell_stock_twice(prices):
    max_total_profit, min_price_so_far = 0.0, float('inf')
    first_buy_sell_profits = [0] * len(prices)
    # Forward phase. For each day, we record maximum profit if we sell on that
    # day.
    for i, price in enumerate(prices):
        min_price_so_far = min(min_price_so_far, price)
        max_total_profit = max(max_total_profit, price - min_price_so_far)
        first_buy_sell_profits[i] = max_total_profit

    # Backward phase. For each day, find the maximum profit if we make the
    # second buy on that day.
    max_price_so_far = float('-inf')
    for i, price in reversed(list(enumerate(prices[1:], 1))):
        max_price_so_far = max(max_price_so_far, price)
        max_total_profit = max(
            max_total_profit,
            max_price_so_far - price + first_buy_sell_profits[i - 1])
    return max_total_profit
```

The time complexity is $O(n)$, and the additional space complexity is $O(n)$, which is the space used to store the best solutions for the subarrays.

Variant: Solve the same problem in $O(n)$ time and $O(1)$ space.

5.8 Computing an alternation

Write a program that takes an array A of n numbers, and rearranges A's elements to get a new array B having the property that $B[0] \leq B[1] \geq B[2] \leq B[3] \geq B[4] \leq B[5] \geq \cdots$.

Hint: Can you solve the problem by making local changes to A?

Solution: One straightforward solution is to sort A and interleave the bottom and top halves of the sorted array. Alternatively, we could sort A and then swap the elements at the pairs $(A[1], A[2]), (A[3], A[4]), \ldots$. Both these approaches have the same time complexity as sorting, namely $O(n \log n)$.

You will soon realize that it is not necessary to sort A to achieve the desired configuration—you could simply rearrange the elements around the median, and then perform the interleaving. Median finding can be performed in time $O(n)$, as per Solution 11.8 on Page 153, which is the overall time complexity of this approach.

Finally, you may notice that the desired ordering is very local, and realize that it is not necessary to find the median. Iterating through the array and swapping $A[i]$ and $A[i+1]$ when i is even and $A[i] > A[i+1]$ or i is odd and $A[i] < A[i+1]$ achieves the desired configuration. In code:

```
def rearrange(A):
```

```
for i in range(len(A)):
    A[i:i + 2] = sorted(A[i:i + 2], reverse=i % 2)
```

This approach has time complexity $O(n)$, which is the same as the approach based on median finding. However, it is much easier to implement and operates in an online fashion, i.e., it never needs to store more than two elements in memory or read a previous element. It nicely illustrates algorithm design by iterative refinement of a brute-force solution.

5.9 Enumerate all primes to n

A natural number is called a prime if it is bigger than 1 and has no divisors other than 1 and itself.

Write a program that takes an integer argument and returns all the primes between 1 and that integer. For example, if the input is 18, you should return $\langle 2, 3, 5, 7, 11, 13, 17 \rangle$.

Hint: Exclude the multiples of primes.

Solution: The natural brute-force algorithm is to iterate over all i from 2 to n, where n is the input to the program. For each i, we test if i is prime; if so we add it to the result. We can use "trial-division" to test if i is prime, i.e., by dividing i by each integer from 2 to the square root of i, and checking if the remainder is 0. (There is no need to test beyond the square root of i, since if i has a divisor other than 1 and itself, it must also have a divisor that is no greater than its square root.) Since each test has time complexity $O(\sqrt{n})$, the time complexity of the entire computation is upper bounded by $O(n \times \sqrt{n})$, i.e., $O(n^{3/2})$.

Intuitively, the brute-force algorithm tests each number from 1 to n independently, and does not exploit the fact that we need to compute *all* primes from 1 to n. Heuristically, a better approach is to compute the primes and when a number is identified as a prime, to "sieve" it, i.e., remove all its multiples from future consideration.

We use a Boolean array to encode the candidates, i.e., if the ith entry in the array is true, then i is potentially a prime. Initially, every number greater than or equal to 2 is a candidate. Whenever we determine a number is a prime, we will add it to the result, which is an array. The first prime is 2. We add it to the result. None of its multiples can be primes, so remove all its multiples from the candidate set by writing false in the corresponding locations. The next location set to true is 3. It must be a prime since nothing smaller than it and greater than 1 is a divisor of it. As before, we add it to result and remove its multiples from the candidate array. We continue till we get to the end of the array of candidates.

As an example, if $n = 10$, the candidate array is initialized to $\langle F, F, T, T, T, T, T, T, T, T \rangle$, where T is true and F is false. (Entries 0 and 1 are false, since 0 and 1 are not primes.) We begin with index 2. Since the corresponding entry is one, we add 2 to the list of primes, and sieve out its multiples. The array is now $\langle F, F, T, T, F, T, F, T, F, T \rangle$. The next nonzero entry is 3, so we add it to the list of primes, and sieve out its multiples. The array is now $\langle F, F, T, T, F, T, F, T, F, F \rangle$. The next nonzero entries are 5 and 7, and neither of them can be used to sieve out more entries.

```
# Given n, return all primes up to and including n.
def generate_primes(n):
    primes = []
    # is_prime[p] represents if p is prime or not. Initially, set each to
    # true, expecting 0 and 1. Then use sieving to eliminate nonprimes.
```

```
    is_prime = [False, False] + [True] * (n - 1)
    for p in range(2, n + 1):
        if is_prime[p]:
            primes.append(p)
            # Sieve p's multiples.
            for i in range(p, n + 1, p):
                is_prime[i] = False
    return primes
```

We justified the sifting approach over the trial-division algorithm on heuristic grounds. The time to sift out the multiples of p is proportional to n/p, so the overall time complexity is $O(n/2 + n/3 + n/5 + n/7 + n/11 + \ldots)$. Although not obvious, this sum asymptotically tends to $n \log\log n$, yielding an $O(n \log\log n)$ time bound. The space complexity is dominated by the storage for P, i.e., $O(n)$.

The bound we gave for the trial-division approach, namely $O(n^{3/2})$, is based on an $O(\sqrt{n})$ bound for each individual test. Since most numbers are not prime, the actual time complexity of trial-division is actually lower on average, since the test frequently early-returns false. It is known that the time complexity of the trial-division approach is $O(n^{3/2}/(\log n)^2)$, so sieving is in fact superior to trial-division.

We can improve runtime by sieving p's multiples from p^2 instead of p, since all numbers of the form kp, where $k < p$ have already been sieved out. The storage can be reduced by ignoring even numbers. The code below reflects these optimizations.

```
# Given n, return all primes up to and including n.
def generate_primes(n):
    if n < 2:
        return []
    size = (n - 3) // 2 + 1
    primes = [2]  # Stores the primes from 1 to n.
    # is_prime[i] represents (2i + 3) is prime or not.
    # Initially set each to true. Then use sieving to eliminate nonprimes.
    is_prime = [True] * size
    for i in range(size):
        if is_prime[i]:
            p = i * 2 + 3
            primes.append(p)
            # Sieving from p^2, where p^2 = (4i^2 + 12i + 9). The index in is_prime
            # is (2i^2 + 6i + 3) because is_prime[i] represents 2i + 3.
            #
            # Note that we need to use long for j because p^2 might overflow.
            for j in range(2 * i**2 + 6 * i + 3, size, p):
                is_prime[j] = False
    return primes
```

The asymptotic time and space complexity are the same as that for the basic sieving approach.

5.10 Permute the elements of an array

A permutation is a rearrangement of members of a sequence into a new sequence. For example, there are 24 permutations of $\langle a, b, c, d\rangle$; some of these are $\langle b, a, d, c\rangle$, $\langle d, a, b, c\rangle$, and $\langle a, d, b, c\rangle$.

A permutation can be specified by an array P, where $P[i]$ represents the location of the element at i in the permutation. For example, the array $\langle 2, 0, 1, 3 \rangle$ represents the permutation that maps the element at location 0 to location 2, the element at location 1 to location 0, the element at location 2 to location 1, and keep the element at location 3 unchanged. A permutation can be applied to an array to reorder the array. For example, the permutation $\langle 2, 0, 1, 3 \rangle$ applied to $A = \langle a, b, c, d \rangle$ yields the array $\langle b, c, a, d \rangle$.

Given an array A of n elements and a permutation P, apply P to A.

Hint: Any permutation can be viewed as a set of cyclic permutations. For an element in a cycle, how would you identify if it has been permuted?

Solution: It is simple to apply a permutation-array to a given array if additional storage is available to write the resulting array. We allocate a new array B of the same length, set $B[P[i]] = A[i]$ for each i, and then copy B to A. The time complexity is $O(n)$, and the additional space complexity is $O(n)$.

A key insight to improving space complexity is to decompose permutations into simpler structures which can be processed incrementally. For example, consider the permutation $\langle 3, 2, 1, 0 \rangle$. To apply it to an array $A = \langle \alpha, b, c, \delta \rangle$, we move the element at index 0 (α) to index 3 and the element already at index 3 (δ) to index 0. Continuing, we move the element at index 1 (b) to index 2 and the element already at index 2 (c) to index 1. Now all elements have been moved according to the permutation, and the result is $\langle \delta, c, b, \alpha \rangle$.

This example generalizes: every permutation can be represented by a collection of independent permutations, each of which is *cyclic*, that is, it moves all elements by a fixed offset, wrapping around.

This is significant, because a single cyclic permutation can be performed one element at a time, i.e., with constant additional storage. Consequently, if the permutation is described as a set of cyclic permutations, it can easily be applied using a constant amount of additional storage by applying each cyclic permutation one-at-a-time. Therefore, we want to identify the disjoint cycles that constitute the permutation.

To find and apply the cycle that includes entry i we just keep going forward (from i to $P[i]$) till we get back to i. After we are done with that cycle, we need to find another cycle that has not yet been applied. It is trivial to do this by storing a Boolean for each array element.

One way to perform this without explicitly using additional $O(n)$ storage is to use the sign bit in the entries in the permutation-array. Specifically, we subtract n from $P[i]$ after applying it. This means that if an entry in $P[i]$ is negative, we have performed the corresponding move.

For example, to apply $\langle 3, 1, 2, 0 \rangle$, we begin with the first entry, 3. We move $A[0]$ to $A[3]$, first saving the original $A[3]$. We update the permutation to $\langle -1, 1, 2, 0 \rangle$. We move $A[3]$ to $A[0]$. Since $P[0]$ is negative we know we are done with the cycle starting at 0. We also update the permutation to $\langle -1, 1, 2, -4 \rangle$. Now we examine $P[1]$. Since it is not negative, it means the cycle it belongs to cannot have been applied. We continue as before.

```
def apply_permutation(perm, A):
    for i in range(len(A)):
        # Check if the element at index i has not been moved by checking if
        # perm[i] is nonnegative.
        next = i
        while perm[next] >= 0:
```

```
            A[i], A[perm[next]] = A[perm[next]], A[i]
            temp = perm[next]
            # Subtracts len(perm) from an entry in perm to make it negative,
            # which indicates the corresponding move has been performed.
            perm[next] -= len(perm)
            next = temp
    # Restore perm.
    perm[:] = [a + len(perm) for a in perm]
```

The program above will apply the permutation in $O(n)$ time. The space complexity is $O(1)$, assuming we can temporarily modify the sign bit from entries in the permutation array.

If we cannot use the sign bit, we can allocate an array of n Booleans indicating whether the element at index i has been processed. Alternatively, we can avoid using $O(n)$ additional storage by going from left-to-right and applying the cycle only if the current position is the leftmost position in the cycle.

```
def apply_permutation(perm, A):
    def cyclic_permutation(start, perm, A):
        i, temp = start, A[start]
        while True:
            next_i = perm[i]
            next_temp = A[next_i]
            A[next_i] = temp
            i, temp = next_i, next_temp
            if i == start:
                break

    for i in range(len(A)):
        # Traverses the cycle to see if i is the minimum element.
        j = perm[i]
        while j != i:
            if j < i:
                break
            j = perm[j]
        else:
            cyclic_permutation(i, perm, A)
```

Testing whether the current position is the leftmost position entails traversing the cycle once more, which increases the run time to $O(n^2)$.

Variant: Given an array A of integers representing a permutation, update A to represent the inverse permutation using only constant additional storage.

5.11 Compute the next permutation

There exist exactly $n!$ permutations of n elements. These can be totally ordered using the *dictionary ordering*—define permutation p to appear before permutation q if in the first place where p and q differ in their array representations, starting from index 0, the corresponding entry for p is less than that for q. For example, $\langle 2,0,1\rangle < \langle 2,1,0\rangle$. Note that the permutation $\langle 0,1,2\rangle$ is the smallest permutation under dictionary ordering, and $\langle 2,1,0\rangle$ is the largest permutation under dictionary ordering.

Write a program that takes as input a permutation, and returns the next permutation under dictionary ordering. If the permutation is the last permutation, return the empty array. For example, if the input is $\langle 1, 0, 3, 2 \rangle$ your function should return $\langle 1, 2, 0, 3 \rangle$. If the input is $\langle 3, 2, 1, 0 \rangle$, return $\langle \rangle$.

Hint: Study concrete examples.

Solution: A brute-force approach might be to find all permutations whose length equals that of the input array, sort them according to the dictionary order, then find the successor of the input permutation in that ordering. Apart from the enormous space and time complexity this entails, simply computing all permutations of length n is a nontrivial problem; see Problem 15.3 on Page 222 for details.

The key insight is that we want to increase the permutation by as little as possible. The loose analogy is how a car's odometer increments; the difference is that we cannot change values, only reorder them. We will use the permutation $\langle 6, 2, 1, 5, 4, 3, 0 \rangle$ to develop this approach.

Specifically, we start from the right, and look at the longest decreasing suffix, which is $\langle 5, 4, 3, 0 \rangle$ for our example. We cannot get the next permutation just by modifying this suffix, since it is already the maximum it can be.

Instead we look at the entry e that appears just before the longest decreasing suffix, which is 1 in this case. (If there's no such element, i.e., the longest decreasing suffix is the entire permutation, the permutation must be $\langle n-1, n-2, \ldots, 2, 1, 0 \rangle$, for which there is no next permutation.)

Observe that e must be less than some entries in the suffix (since the entry immediately after e is greater than e). Intuitively, we should swap e with the smallest entry s in the suffix which is larger than e so as to minimize the change to the prefix (which is defined to be the part of the sequence that appears before the suffix).

For our example, e is 1 and s is 3. Swapping s and e results in $\langle 6, 2, 3, 5, 4, 1, 0 \rangle$.

We are not done yet—the new prefix is the smallest possible for all permutations greater than the initial permutation, but the new suffix may not be the smallest. We can get the smallest suffix by sorting the entries in the suffix from smallest to largest. For our working example, this yields the suffix $\langle 0, 1, 4, 5 \rangle$.

As an optimization, it is not necessary to call a full blown sorting algorithm on suffix. Since the suffix was initially decreasing, and after replacing s by e it remains decreasing, reversing the suffix has the effect of sorting it from smallest to largest.

The general algorithm for computing the next permutation is as follows:

(1.) Find k such that $p[k] < p[k+1]$ and entries after index k appear in decreasing order.

(2.) Find the smallest $p[l]$ such that $p[l] > p[k]$ (such an l must exist since $p[k] < p[k+1]$).

(3.) Swap $p[l]$ and $p[k]$ (note that the sequence after position k remains in decreasing order).

(4.) Reverse the sequence after position k.

```
def next_permutation(perm):
    # Find the first entry from the right that is smaller than the entry
    # immediately after it.
    inversion_point = len(perm) - 2
    while (inversion_point >= 0
           and perm[inversion_point] >= perm[inversion_point + 1]):
        inversion_point -= 1
    if inversion_point == -1:
        return []  # perm is the last permutation.
```

```
    # Swap the smallest entry after index inversion_point that is greater than
    # perm[inversion_point]. Since entries in perm are decreasing after
    # inversion_point, if we search in reverse order, the first entry that is
    # greater than perm[inversion_point] is the entry to swap with.
    for i in reversed(range(inversion_point + 1, len(perm))):
        if perm[i] > perm[inversion_point]:
            perm[inversion_point], perm[i] = perm[i], perm[inversion_point]
            break

    # Entries in perm must appear in decreasing order after inversion_point,
    # so we simply reverse these entries to get the smallest dictionary order.
    perm[inversion_point + 1:] = reversed(perm[inversion_point + 1:])
    return perm
```

Each step is an iteration through an array, so the time complexity is $O(n)$. All that we use are a few local variables, so the additional space complexity is $O(1)$.

Variant: Compute the kth permutation under dictionary ordering, starting from the identity permutation (which is the first permutation in dictionary ordering).

Variant: Given a permutation p, return the permutation corresponding to the *previous* permutation of p under dictionary ordering.

5.12 Sample offline data

This problem is motivated by the need for a company to select a random subset of its customers to roll out a new feature to. For example, a social networking company may want to see the effect of a new UI on page visit duration without taking the chance of alienating all its users if the rollout is unsuccessful.

Implement an algorithm that takes as input an array of distinct elements and a size, and returns a subset of the given size of the array elements. All subsets should be equally likely. Return the result in input array itself.

Hint: How would you construct a random subset of size $k+1$ given a random subset of size k?

Solution: Let the input array be A, its length n, and the specified size k. A naive approach is to iterate through the input array, selecting entries with probability k/n. Although the average number of selected entries is k, we may select more or less than k entries in this way.

Another approach is to enumerate all subsets of size k and then select one at random from these. Since there are $\binom{n}{k}$ subsets of size k, the time and space complexity are huge. Furthermore, enumerating all subsets of size k is nontrivial (Problem 15.5 on Page 226).

The key to efficiently building a random subset of size exactly k is to first build one of size $k-1$ and then adding one more element, selected randomly from the rest. The problem is trivial when $k = 1$. We make one call to the random number generator, take the returned value mod n (call it r), and swap $A[0]$ with $A[r]$. The entry $A[0]$ now holds the result.

For $k > 1$, we begin by choosing one element at random as above and we now repeat the same process with the $n-1$ element subarray $A[1, n-1]$. Eventually, the random subset occupies the slots $A[0, k-1]$ and the remaining elements are in the last $n-k$ slots.

Intuitively, if all subsets of size k are equally likely, then the construction process ensures that the subsets of size $k+1$ are also equally likely. A formal proof, which we do not present, uses mathematical induction—the induction hypothesis is that every permutation of every size k subset of A is equally likely to be in $A[0, k-1]$.

As a concrete example, let the input be $A = \langle 3, 7, 5, 11 \rangle$ and the size be 3. In the first iteration, we use the random number generator to pick a random integer in the interval $[0, 3]$. Let the returned random number be 2. We swap $A[0]$ with $A[2]$—now the array is $\langle 5, 7, 3, 11 \rangle$. Now we pick a random integer in the interval $[1, 3]$. Let the returned random number be 3. We swap $A[1]$ with $A[3]$—now the resulting array is $\langle 5, 11, 3, 7 \rangle$. Now we pick a random integer in the interval $[2, 3]$. Let the returned random number be 2. When we swap $A[2]$ with itself the resulting array is unchanged. The random subset consists of the first three entries, i.e., $\{5, 11, 3\}$.

```
def random_sampling(k, A):
    for i in range(k):
        # Generate a random index in [i, len(A) - 1].
        r = random.randint(i, len(A) - 1)
        A[i], A[r] = A[r], A[i]
```

The algorithm clearly runs in additional $O(1)$ space. The time complexity is $O(k)$ to select the elements.

The algorithm makes k calls to the random number generator. When k is bigger than $\frac{n}{2}$, we can optimize by computing a subset of $n-k$ elements to remove from the set. For example, when $k = n-1$, this replaces $n-1$ calls to the random number generator with a single call.

Variant: The `rand()` function in the standard C library returns a uniformly random number in $[0, \texttt{RAND\_MAX} - 1]$. Does `rand()` mod n generate a number uniformly distributed in $[0, n-1]$?

5.13 Sample online data

This problem is motivated by the design of a packet sniffer that provides a uniform sample of packets for a network session.

Design a program that takes as input a size k, and reads packets, continuously maintaining a uniform random subset of size k of the read packets.

Hint: Suppose you have a procedure which selects k packets from the first $n \geq k$ packets as specified. How would you deal with the $(n+1)$th packet?

Solution: A brute force approach would be to store all the packets read so far. After reading in each packet, we apply Solution 5.12 on the preceding page to compute a random subset of k packets. The space complexity is high—$O(n)$, after n packets have been read. The time complexity is also high—$O(nk)$, since each packet read is followed by a call to Solution 5.12 on the facing page.

At first glance it may seem that it is impossible to do better than the brute-force approach, since after reading the nth packet, we need to choose k packets uniformly from this set. However, suppose we have read the first n packets, and have a random subset of k of them. When we read the $(n+1)$th packet, it should belong to the new subset with probability $k/(n+1)$. If we choose one of the packets in the existing subset uniformly randomly to remove, the resulting collection will be a random subset of the $n+1$ packets.

The formal proof that the algorithm works correctly, uses induction on the number of packets that have been read. Specifically, the induction hypothesis is that all k-sized subsets are equally likely after $n \geq k$ packets have been read.

As an example, suppose $k = 2$, and the packets are read in the order p, q, r, t, u, v. We keep the first two packets in the subset, which is $\{p, q\}$. We select the next packet, r, with probability 2/3. Suppose it is not selected. Then the subset after reading the first three packets is still $\{p, q\}$. We select the next packet, t, with probability 2/4. Suppose it is selected. Then we choose one of the packets in $\{p, q\}$ uniformly, and replace it with t. Let q be the selected packet—now the subset is $\{p, t\}$. We select the next packet u with probability 2/5. Suppose it is selected. Then we choose one of the packets in $\{p, t\}$ uniformly, and replace it with u. Let t be the selected packet—now the subset is $\{p, u\}$. We select the next packet v with probability 2/6. Suppose it is not selected. The random subset remains $\{p, u\}$.

```
# Assumption: there are at least k elements in the stream.
def online_random_sample(it, k):
    # Stores the first k elements.
    sampling_results = list(itertools.islice(it, k))

    # Have read the first k elements.
    num_seen_so_far = k
    for x in it:
        num_seen_so_far += 1
        # Generate a random number in [0, num_seen_so_far - 1], and if this
        # number is in [0, k - 1], we replace that element from the sample with
        # x.
        idx_to_replace = random.randrange(num_seen_so_far)
        if idx_to_replace < k:
            sampling_results[idx_to_replace] = x
    return sampling_results
```

The time complexity is proportional to the number of elements in the stream, since we spend $O(1)$ time per element. The space complexity is $O(k)$.

Note that at each iteration, every subset is equally likely. However, the subsets are not independent from iteration to iteration—successive subsets differ in at most one element. In contrast, the subsets computed by brute-force algorithm are independent from iteration to iteration.

5.14 Compute a random permutation

Generating random permutations is not as straightforward as it seems. For example, iterating through $\langle 0, 1, \ldots, n-1 \rangle$ and swapping each element with another randomly selected element does *not* generate all permutations with equal probability. One way to see this is to consider the case $n = 3$. The number of permutations is $3! = 6$. The total number of ways in which we can choose the elements to swap is $3^3 = 27$ and all are equally likely. Since 27 is not divisible by 6, some permutations correspond to more ways than others, so not all permutations are equally likely.

Design an algorithm that creates uniformly random permutations of $\{0, 1, \ldots, n-1\}$. You are given a random number generator that returns integers in the set $\{0, 1, \ldots, n-1\}$ with equal probability; use as few calls to it as possible.

Hint: If the result is stored in A, how would you proceed once $A[n-1]$ is assigned correctly?

Solution: A brute-force approach might be to iteratively pick random numbers between 0 and $n-1$, inclusive. If number repeats, we discard it, and try again. A hash table is a good way to store and test values that have already been picked.

For example, if $n = 4$, we might have the sequence 1, 2, 1 (repeats), 3, 1 (repeats), 2 (repeat), 0 (done, all numbers from 0 to 3 are present). The corresponding permutation is $\langle 1, 2, 3, 0 \rangle$.

It is fairly clear that all permutations are equally likely with this approach. The space complexity beyond that of the result array is $O(n)$ for the hash table. The time complexity is slightly challenging to analyze. Early on, it takes very few iterations to get more new values, but it takes a long time to collect the last few values. Computing the average number of tries to complete the permutation in this way is known as the Coupon Collector's Problem. It is known that the number of tries on average (and hence the average time complexity) is $O(n \log n)$.

Clearly, the way to improve time complexity is to avoid repeats. We can do this by restricting the set we randomly choose the remaining values from. If we apply Solution 5.12 on Page 54 to $\langle 0, 1, 2, \ldots, n-1 \rangle$ with $k = n$, at each iteration the array is partitioned into the partial permutation and remaining values. Although the subset that is returned is unique (it will be $\{0, 1, \ldots, n-1\}$), all $n!$ possible orderings of the elements in the set occur with equal probability. For example, let $n = 4$. We begin with $\langle 0, 1, 2, 3 \rangle$. The first random number is chosen between 0 and 3, inclusive. Suppose it is 1. We update the array to $\langle 1, 0, 2, 3 \rangle$. The second random number is chosen between 1 and 3, inclusive. Suppose it is 3. We update the array to $\langle 1, 3, 0, 2 \rangle$. The third random number is chosen between 2 and 3, inclusive. Suppose it is 3. We update the array to $\langle 1, 3, 2, 0 \rangle$. This is the returned result.

```
def compute_random_permutation(n):
    permutation = list(range(n))
    random_sampling(n, permutation)
    return permutation
```

The time complexity is $O(n)$, and, as an added bonus, no storage outside of that needed for the permutation array itself is needed.

5.15 Compute a random subset

The set $\{0, 1, 2, \ldots, n-1\}$ has $\binom{n}{k} = n!/((n-k)!k!)$ subsets of size k. We seek to design an algorithm that returns any one of these subsets with equal probability.

Write a program that takes as input a positive integer n and a size $k \leq n$, and returns a size-k subset of $\{0, 1, 2, \ldots, n-1\}$. The subset should be represented as an array. All subsets should be equally likely and, in addition, all permutations of elements of the array should be equally likely. You may assume you have a function which takes as input a nonnegative integer t and returns an integer in the set $\{0, 1, \ldots, t-1\}$ with uniform probability.

Hint: Simulate Solution 5.12 on Page 54, using an appropriate data structure to reduce space.

Solution: Similar to the brute-force algorithm presented in Solution 5.14 on the preceding page, we could iteratively choose random numbers between 0 and $n-1$ until we get k distinct values. This

approach suffers from the same performance degradation when k is close to n, and it also requires $O(k)$ additional space.

We could mimic the offline sampling algorithm described in Solution 5.12 on Page 54, with $A[i] = i$ initially, stopping after k iterations. This requires $O(n)$ space and $O(n)$ time to create the array. After creating $\langle 0, 1, 2, \ldots, n-1 \rangle$, we need $O(k)$ time to produce the subset.

Note that when $k \ll n$, most of the array is untouched, i.e., $A[i] = i$. The key to reducing the space complexity to $O(k)$ is simulating A with a hash table. We do this by only tracking entries whose values are modified by the algorithm—the remainder have the default value, i.e., the value of an entry is its index.

Specifically, we maintain a hash table H whose keys and values are from $\{0, 1, \ldots, n-1\}$. Conceptually, H tracks entries of the array which have been touched in the process of randomization—these are entries $A[i]$ which may not equal i. The hash table H is updated as the algorithm advances.

- If i is in H, then its value in H is the value stored at $A[i]$ in the brute-force algorithm.
- If i is not in H, then this implicitly implies $A[i] = i$.

Since we track no more than k entries, when k is small compared to n, we save time and space over the brute-force approach, which has to initialize and update an array of length n.

Initially, H is empty. We do k iterations of the following. Choose a random integer r in $[0, n-1-i]$, where i is the current iteration count, starting at 0. There are four possibilities, corresponding to whether the two entries in A that are being swapped are already present or not present in H. The desired result is in $A[0, k-1]$, which can be determined from H.

For example, suppose $n = 100$ and $k = 4$. In the first iteration, suppose we get the random number 28. We update H to $(0, 28), (28, 0)$. This means that $A[0]$ is 28 and $A[28]$ is 0—for all other i, $A[i] = i$. In the second iteration, suppose we get the random number 42. We update H to $(0, 28), (28, 0), (1, 42), (42, 1)$. In the third iteration, suppose we get the random number 28 again. We update H to $(0, 28), (28, 2), (1, 42), (42, 1), (2, 0)$. In the third iteration, suppose we get the random number 64. We update H to $(0, 28), (28, 2), (1, 42), (42, 1), (2, 0), (3, 64), (64, 3)$. The random subset is the 4 elements corresponding to indices $0, 1, 2, 3$, i.e., $\langle 28, 42, 0, 64 \rangle$.

```
def random_subset(n, k):
    changed_elements = {}
    for i in range(k):
        # Generate a random index between i and n - 1, inclusive.
        rand_idx = random.randrange(i, n)
        rand_idx_mapped = changed_elements.get(rand_idx, rand_idx)
        i_mapped = changed_elements.get(i, i)
        changed_elements[rand_idx] = i_mapped
        changed_elements[i] = rand_idx_mapped
    return [changed_elements[i] for i in range(k)]
```

The time complexity is $O(k)$, since we perform a bounded number of operations per iteration. The space complexity is also $O(k)$, since H and the result array never contain more than k entries.

5.16 Generate nonuniform random numbers

Suppose you need to write a load test for a server. You have studied the inter-arrival time of requests to the server over a period of one year. From this data you have computed a histogram

of the distribution of the inter-arrival time of requests. In the load test you would like to generate requests for the server such that the inter-arrival times come from the same distribution that was observed in the historical data. The following problem formalizes the generation of inter-arrival times.

You are given n numbers as well as probabilities $p_0, p_1, \ldots, p_{n-1}$, which sum up to 1. Given a random number generator that produces values in $[0, 1)$ uniformly, how would you generate one of the n numbers according to the specified probabilities? For example, if the numbers are $3, 5, 7, 11$, and the probabilities are $9/18, 6/18, 2/18, 1/18$, then in 1000000 calls to your program, 3 should appear roughly 500000 times, 5 should appear roughly 333333 times, 7 should appear roughly 111111 times, and 11 should appear roughly 55555 times.

Hint: Look at the graph of the probability that the selected number is less than or equal to α. What do the jumps correspond to?

Solution: First note that actual values of the numbers is immaterial—we want to choose from one of n outcomes with probabilities $p_0, p_1, \ldots, p_{n-1}$. If all probabilities were the same, i.e., $1/n$, we could make a single call to the random number generator, and choose outcome i if the number falls lies between i/n and $(i+1)/n$.

For the case where the probabilities are not the same, we can solve the problem by partitioning the unit interval $[0, 1]$ into n disjoint segments, in a way so that the length of the jth interval is proportional to p_j. Then we select a number uniformly at random in the unit interval, $[0, 1]$, and return the number corresponding to the interval the randomly generated number falls in.

An easy way to create these intervals is to use $p_0, p_0 + p_1, p_0 + p_1 + p_2, \ldots, p_0 + p_1 + p_2 + \cdots + p_{n-1}$ as the endpoints. Using the example given in the problem statement, the four intervals are $[0.0, 0.5), [0.5, 0.833), [0.833, 0.944), [0.944, 1.0]$. Now, for example, if the random number generated uniformly in $[0.0, 1.0]$ is 0.873, since 0.873 lies in $[0.833, 0.944)$, which is the third interval, we return the third number, which is 7.

In general, searching an array of n disjoint intervals for the interval containing a number takes $O(n)$ time. However, we can do better. Since the array $\langle p_0, p_0+p_1, p_0+p_1+p_2, \ldots, p_0+p_1+p_2+\cdots+p_{n-1}\rangle$ is sorted, we can use binary search to find the interval in $O(\log n)$ time.

```
def nonuniform_random_number_generation(values, probabilities):
    prefix_sum_of_probabilities = list(itertools.accumulate(probabilities))
    interval_idx = bisect.bisect(prefix_sum_of_probabilities, random.random())
    return values[interval_idx]
```

The time complexity to compute a single value is $O(n)$, which is the time to create the array of intervals. This array also implies an $O(n)$ space complexity.

Once the array is constructed, computing each additional result entails one call to the uniform random number generator, followed by a binary search, i.e., $O(\log n)$.

Variant: Given a random number generator that produces values in $[0, 1]$ uniformly, how would you generate a value X from T according to a continuous probability distribution, such as the exponential distribution?

Multidimensional arrays

Thus far we have focused our attention in this chapter on one-dimensional arrays. We now turn our attention to multidimensional arrays. A 2D array in an array whose entries are themselves arrays; the concept generalizes naturally to k dimensional arrays.

Multidimensional arrays arise in image processing, board games, graphs, modeling spatial phenomenon, etc. Often, but not always, the arrays that constitute the entries of a 2D array A have the same length, in which case we refer to A as being an $m \times n$ rectangular array (or sometimes just an $m \times n$ array), where m is the number of entries in A, and n the number of entries in $A[0]$. The elements within a 2D array A are often referred to by their *row* and *column* indices i and j, and written as $A[i][j]$.

5.17 The Sudoku checker problem

Sudoku is a popular logic-based combinatorial number placement puzzle. The objective is to fill a 9×9 grid with digits subject to the constraint that each column, each row, and each of the nine 3×3 sub-grids that compose the grid contains unique integers in $[1, 9]$. The grid is initialized with a partial assignment as shown in Figure 5.2(a); a complete solution is shown in Figure 5.2(b).

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

(a) Partial assignment.

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

(b) A complete solution.

Figure 5.2: Sudoku configurations.

Check whether a 9×9 2D array representing a partially completed Sudoku is valid. Specifically, check that no row, column, or 3×3 2D subarray contains duplicates. A 0-value in the 2D array indicates that entry is blank; every other entry is in $[1, 9]$.

Hint: Directly test the constraints. Use an array to encode sets.

Solution: There is no real scope for algorithm optimization in this problem—it's all about writing clean code.

We need to check nine row constraints, nine column constraints, and nine sub-grid constraints. It is convenient to use bit arrays to test for constraint violations, that is to ensure no number in $[1, 9]$ appears more than once.

```
# Check if a partially filled matrix has any conflicts.
```

```
def is_valid_sudoku(partial_assignment):
    # Return True if subarray
    # partial_assignment[start_row:end_row][start_col:end_col] contains any
    # duplicates in {1, 2, ..., len(partial_assignment)}; otherwise return
    # False.
    def has_duplicate(block):
        block = list(filter(lambda x: x != 0, block))
        return len(block) != len(set(block))

    n = len(partial_assignment)
    # Check row and column constraints.
    if any(
            has_duplicate([partial_assignment[i][j] for j in range(n)])
            or has_duplicate([partial_assignment[j][i] for j in range(n)])
            for i in range(n)):
        return False

    # Check region constraints.
    region_size = int(math.sqrt(n))
    return all(not has_duplicate([
        partial_assignment[a][b]
        for a in range(region_size * I, region_size * (I + 1))
        for b in range(region_size * J, region_size * (J + 1))
    ]) for I in range(region_size) for J in range(region_size))


# Pythonic solution that exploits the power of list comprehension.
def is_valid_sudoku_pythonic(partial_assignment):
    region_size = int(math.sqrt(len(partial_assignment)))
    return max(
        collections.Counter(k
                            for i, row in enumerate(partial_assignment)
                            for j, c in enumerate(row)
                            if c != 0
                            for k in ((i, str(c)), (str(c), j),
                                      (i / region_size, j / region_size,
                                       str(c)))).values(),
        default=0) <= 1
```

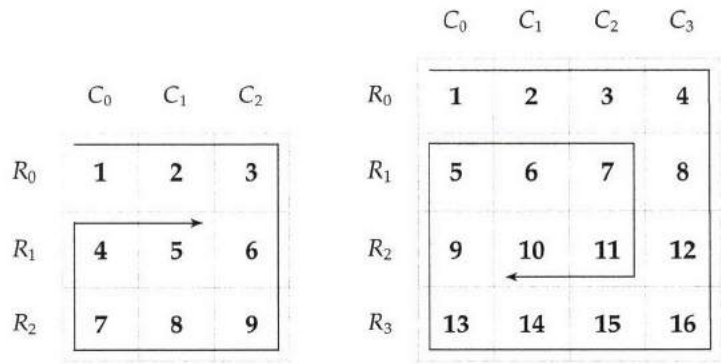
The time complexity of this algorithm for an $n \times n$ Sudoku grid with $\sqrt{n} \times \sqrt{n}$ subgrids is $O(n^2) + O(n^2) + O(n^2/(\sqrt{n})^2 \times (\sqrt{n})^2) = O(n^2)$; the terms correspond to the complexity to check n row constraints, the n column constraints, and the n subgrid constraints, respectively. The memory usage is dominated by the bit array used to check the constraints, so the space complexity is $O(n)$.

Solution 15.9 on Page 230 describes how to solve Sudoku instances.

5.18 Compute the spiral ordering of a 2D array

A 2D array can be written as a sequence in several orders—the most natural ones being row-by-row or column-by-column. In this problem we explore the problem of writing the 2D array in spiral order. For example, the spiral ordering for the 2D array in Figure 5.3(a) on the fol-

lowing page is $\langle 1,2,3,6,9,8,7,4,5\rangle$. For Figure 5.3(b) on the next page, the spiral ordering is $\langle 1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10\rangle$.



(a) Spiral ordering for a 3×3 array. (b) Spiral ordering for a 4 × 4 array.

Figure 5.3: Spiral orderings. Column and row ids are specified above and to the left of the matrix. The value 1 is at entry R_0, C_0.

Write a program which takes an $n \times n$ 2D array and returns the spiral ordering of the array.

Hint: Use case analysis and divide-and-conquer.

Solution: It is natural to solve this problem starting from the outside, and working to the center. The naive approach begins by adding the first row, which consists of n elements. Next we add the $n-1$ remaining elements of the last column, then the $n-1$ remaining elements of the last row, and then the $n-2$ remaining elements of the first column. The lack of uniformity makes it hard to get the code right.

Here is a uniform way of adding the boundary. Add the first $n-1$ elements of the first row. Then add the first $n-1$ elements of the last column. Then add the last $n-1$ elements of the last row in reverse order. Finally, add the last $n-1$ elements of the first column in reverse order.

After this, we are left with the problem of adding the elements of an $(n-2) \times (n-2)$ 2D array in spiral order. This leads to an iterative algorithm that adds the outermost elements of $n \times n, (n-2) \times (n-2), (n-4) \times (n-4), \ldots$ 2D arrays. Note that a matrix of odd dimension has a corner-case, namely when we reach its center.

As an example, for the 3×3 array in Figure 5.3(a), we would add $1,2$ (first two elements of the first row), then $3,6$ (first two elements of the last column), then $9,8$ (last two elements of the last row), then $7,4$ (last two elements of the first column). We are now left with the 1×1 array, whose sole element is 5. After processing it, all elements are processed.

For the 4×4 array in Figure 5.3(b), we would add $1,2,3$ (first three elements of the first row), then $4,8,12$ (first three elements of the last column), then $16,15,14$ (last three elements of the last row), then $13,9,5$ (last three elements of the first column). We are now left with a 2×2 matrix, which we process similarly in the order $6,7,11,10$, after which all elements are processed.

```
def matrix_in_spiral_order(square_matrix):
    def matrix_layer_in_clockwise(offset):
        if offset == len(square_matrix) - offset - 1:
            # square_matrix has odd dimension, and we are at the center of the
```

```
            # matrix square_matrix.
            spiral_ordering.append(square_matrix[offset][offset])
            return

        spiral_ordering.extend(square_matrix[offset][offset:-1 - offset])
        spiral_ordering.extend(
            list(zip(*square_matrix))[-1 - offset][offset:-1 - offset])
        spiral_ordering.extend(
            square_matrix[-1 - offset][-1 - offset:offset:-1])
        spiral_ordering.extend(
            list(zip(*square_matrix))[offset][-1 - offset:offset:-1])

    spiral_ordering = []
    for offset in range((len(square_matrix) + 1) // 2):
        matrix_layer_in_clockwise(offset)
    return spiral_ordering
```

The time complexity is $O(n^2)$.

The above solution uses four iterations which are almost identical. Now we present a solution that uses a single iteration. We start at the entry at R_0, C_0. We process entries in the sequence $R_0, C_0, R_0, C_1, \ldots, R_0, C_{n-2}$, i.e., we are moving to the right. Then we process entries $R_0, C_{n-1}, R_1, C_{n-1}, \ldots, R_{n-2}, C_{n-1}$, i.e., we are moving down. Then we process entries $R_{n-1}, C_{n-1}, R_{n-1}, C_{n-2}, \ldots, R_{n-1}, C_1$, i.e., we are moving to the left. Then we process entries $R_{n-1}, C_0, R_{n-2}, C_0, \ldots, R_1, C_0$, i.e., we are moving up. We record that an element has already been processed by setting it to 0, which is assumed to be a value that is not already present in the array. (Any value not in the array works too.) After processing the entry R_1, C_0 we start the same process from R_1, C_1. This method is applied until all elements are processed. Conceptually, we are processing the array in "shells" from the outside moving to the center.

```
def matrix_in_spiral_order(square_matrix):
    SHIFT = ((0, 1), (1, 0), (0, -1), (-1, 0))
    direction = x = y = 0
    spiral_ordering = []

    for _ in range(len(square_matrix)**2):
        spiral_ordering.append(square_matrix[x][y])
        square_matrix[x][y] = 0
        next_x, next_y = x + SHIFT[direction][0], y + SHIFT[direction][1]
        if (next_x not in range(len(square_matrix))
                or next_y not in range(len(square_matrix))
                or square_matrix[next_x][next_y] == 0):
            direction = (direction + 1) & 3
            next_x, next_y = x + SHIFT[direction][0], y + SHIFT[direction][1]
        x, y = next_x, next_y
    return spiral_ordering
```

The time complexity is $O(n^2)$ and the space complexity is $O(1)$.

Variant: Given a dimension d, write a program to generate a $d \times d$ 2D array which in spiral order is

$\langle 1, 2, 3, \ldots, d^2 \rangle$. For example, if $d = 3$, the result should be

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 8 & 9 & 4 \\ 7 & 6 & 5 \end{bmatrix}.$$

Variant: Given a sequence of integers P, compute a 2D array A whose spiral order is P. (Assume the size of P is n^2 for some integer n.)

Variant: Write a program to enumerate the first n pairs of integers (a, b) in spiral order, starting from $(0,0)$ followed by $(1,0)$. For example, if $n = 10$, your output should be $(0,0), (1,0), (1,-1), (0,-1), (-1,-1), (-1,0), (-1,1), (0,1), (1,1), (2,1)$.

Variant: Compute the spiral order for an $m \times n$ 2D array A.

Variant: Compute the last element in spiral order for an $m \times n$ 2D array A in $O(1)$ time.

Variant: Compute the kth element in spiral order for an $m \times n$ 2D array A in $O(1)$ time.

5.19 Rotate a 2D array

Image rotation is a fundamental operation in computer graphics. Figure 5.4 illustrates the rotation operation on a 2D array representing a bit-map of an image. Specifically, the image is rotated by 90 degrees clockwise.

1	**2**	**3**	**4**
5	**6**	**7**	**8**
9	**10**	**11**	**12**
13	**14**	**15**	**16**

(a) Initial 4×4 2D array.

13	**9**	**5**	**1**
14	**10**	**6**	**2**
15	**11**	**7**	**3**
16	**12**	**8**	**4**

(b) Array rotated by 90 degrees clockwise.

Figure 5.4: Example of 2D array rotation.

Write a function that takes as input an $n \times n$ 2D array, and rotates the array by 90 degrees clockwise.

Hint: Focus on the boundary elements.

Solution: With a little experimentation, it is easy to see that ith column of the rotated matrix is the ith row of the original matrix. For example, the first row, $\langle 13, 14, 15, 16 \rangle$ of the initial array in 5.4 becomes the first column in the rotated version. Therefore, a brute-force approach is to allocate a new $n \times n$ 2D array, write the rotation to it (writing rows of the original matrix into the columns of the new matrix), and then copying the new array back to the original one. The last step is needed since the problem says to update the original array. The time and additional space complexity are both $O(n^2)$.

Since we are not explicitly required to allocate a new array, it is natural to ask if we can perform the rotation in-place, i.e., with $O(1)$ additional storage. The first insight is that we can perform the rotation in a layer-by-layer fashion—different layers can be processed independently. Furthermore, within a layer, we can exchange groups of four elements at a time to perform the rotation, e.g., send 1 to 4's location, 4 to 16's location, 16 to 13's location, and 13 to 1's location, then send 2 to 8's location, 8 to 15's location, 15 to 9's location, and 9 to 2's location, etc. The program below works its way into the center of the array from the outermost layers, performing exchanges within a layer iteratively using the four-way swap just described.

```
def rotate_matrix(square_matrix):
    matrix_size = len(square_matrix) - 1
    for i in range(len(square_matrix) // 2):
        for j in range(i, matrix_size - i):
            # Perform a 4-way exchange. Note that A[~i] for i in [0, len(A) - 1]
            # is A[-(i + 1)].
            (square_matrix[i][j], square_matrix[~j][i], square_matrix[~i][~j],
             square_matrix[j][~i]) = (square_matrix[~j][i],
                                      square_matrix[~i][~j],
                                      square_matrix[j][~i], square_matrix[i][j])
```

The time complexity is $O(n^2)$ and the additional space complexity is $O(1)$.

Interestingly, we can get the effect of a rotation with $O(1)$ space and time complexity, albeit with some limitations. Specifically, we return an object r that composes the original matrix A. A read of the element at indices i and j in r is converted into a read from A at index $[n-1-j][i]$. Writes are handled similarly. The time to create r is constant, since it simply consists of a reference to A. The time to perform reads and writes is unchanged. This approach breaks when there are clients of the original A object, since writes to r change A. Even if A is not written to, if methods on the stored objects change their state, the system gets corrupted. Copy-on-write can be used to solve these issues.

```
class RotatedMatrix:
    def __init__(self, square_matrix):
        self._square_matrix = square_matrix

    def read_entry(self, i, j):
        # Note that A[~i] for i in [0, len(A) - 1] is A[~(i + 1)].
        return self._square_matrix[~j][i]

    def write_entry(self, i, j, v):
        self._square_matrix[~j][i] = v
```

Variant: Implement an algorithm to reflect A, assumed to be an $n \times n$ 2D array, about the horizontal axis of symmetry. Repeat the same for reflections about the vertical axis, the diagonal from top-left to bottom-right, and the diagonal from top-right to bottom-left.

5.20 Compute rows in Pascal's Triangle

Figure 5.5 on the following page shows the first five rows of a graphic that is known as Pascal's triangle. Each row contains one more entry than the previous one. Except for entries in the last

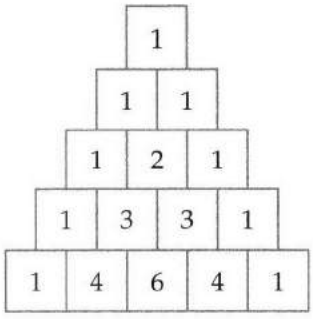


Figure 5.5: A Pascal triangle.

row, each entry is adjacent to one or two numbers in the row below it. The first row holds 1. Each entry holds the sum of the numbers in the adjacent entries above it.

Write a program which takes as input a nonnegative integer n and returns the first n rows of Pascal's triangle.

Hint: Write the given fact as an equation.

Solution: A brute-force approach might be to organize the arrays in memory similar to how they appear in the figure. The challenge is to determine the correct indices to range over and to read from.

A better approach is to keep the arrays left-aligned, that is the first entry is at location 0. Now it is simple: the jth entry in the ith row is 1 if $j = 0$ or $j = i$, otherwise it is the sum of the $(j-1)$th and jth entries in the $(i-1)$th row. The first row R_0 is $\langle 1 \rangle$. The second row R_1 is $\langle 1, 1 \rangle$. The third row R_2 is $\langle 1, R_1[0] + R_1[1] = 2, 1 \rangle$. The fourth row R_3 is $\langle 1, R_2[0] + R_2[1] = 3, R_2[1] + R_2[2] = 3, 1 \rangle$.

```
def generate_pascal_triangle(n):
    result = [[1] * (i + 1) for i in range(n)]
    for i in range(n):
        for j in range(1, i):
            # Sets this entry to the sum of the two above adjacent entries.
            result[i][j] = result[i - 1][j - 1] + result[i - 1][j]
    return result
```

Since each element takes $O(1)$ time to compute, the time complexity is $O(1 + 2 + \cdots + n) = O(n(n+1)/2) = O(n^2)$. Similarly, the space complexity is $O(n^2)$.

It is a fact that the ith entry in the nth row of Pascal's triangle is $\binom{n}{i}$. This in itself does not trivialize the problem, since computing $\binom{n}{i}$ itself is tricky. (In fact, Pascal's triangle can be used to compute $\binom{n}{i}$.)

Variant: Compute the nth row of Pascal's triangle using $O(n)$ space.

CHAPTER

6 Strings

String pattern matching is an important problem that occurs in many areas of science and information processing. In computing, it occurs naturally as part of data processing, text editing, term rewriting, lexical analysis, and information retrieval.

— *"Algorithms For Finding Patterns in Strings,"*
A. V. AHO, 1990

Strings are ubiquitous in programming today—scripting, web development, and bioinformatics all make extensive use of strings.

A string can be viewed as a special kind of array, namely one made out of characters. We treat strings separately from arrays because certain operations which are commonly applied to strings—for example, comparison, joining, splitting, searching for substrings, replacing one string by another, parsing, etc.—do not make sense for general arrays.

You should know how strings are represented in memory, and understand basic operations on strings such as comparison, copying, joining, splitting, matching, etc. Advanced string processing algorithms often use hash tables (Chapter 12) and dynamic programming (Page 234). In this chapter we present problems on strings which can be solved using basic techniques.

Strings boot camp

A palindromic string is one which reads the same when it is reversed. The program below checks whether a string is palindromic. Rather than creating a new string for the reverse of the input string, it traverses the input string forwards and backwards, thereby saving space. Notice how it uniformly handles even and odd length strings.

```
def is_palindromic(s):
    # Note that s[~i] for i in [0, len(s) - 1] is s[-(i + 1)].
    return all(s[i] == s[~i] for i in range(len(s) // 2))
```

The time complexity is $O(n)$ and the space complexity is $O(1)$, where n is the length of the string.

Know your string libraries

The key operators and functions on strings are `s[3]`, `len(s)`, `s + t`, `s[2:4]` (and all the other variants of slicing for lists described on Page 39), `s in t`, `s.strip()`, `s.startswith(prefix)`, `s.endswith(suffix)`, `'Euclid,Axiom 5,Parallel Lines'.split(',')`, `3 * '01'`, `','.join(('Gauss', 'Prince of Mathematicians', '1777-1855'))`, `s.tolower()`, and `'Name {name}, Rank {rank}'.format(name='Archimedes', rank=3)`.

It's important to remember that strings are immutable—operations like `s = s[1:]` or `s += '123'` imply creating a new array of characters that is then assigned back to `s`. This implies that concatenating a single character n times to a string in a for loop has $O(n^2)$ time complexity. (Some

Similar to arrays, string problems often have simple brute-force solutions that use $O(n)$ space solution, but subtler solutions that use the string itself to **reduce space complexity** to $O(1)$. Understand the **implications** of a string type which is **immutable**, e.g., the need to allocate a new string when concatenating immutable strings. Know **alternatives** to immutable strings, e.g., a `list` in Python. Updating a mutable string from the front is slow, so see if it's possible to **write values from the back**.

Table 6.1: Top Tips for Strings

implementations of Python use tricks under-the-hood to avoid having to do this allocation, reducing the complexity to $O(n)$.)

6.1 Interconvert strings and integers

A string is a sequence of characters. A string may encode an integer, e.g., "123" encodes 123. In this problem, you are to implement methods that take a string representing an integer and return the corresponding integer, and vice versa. Your code should handle negative integers. You cannot use library functions like `int` in Python.

Implement an integer to string conversion function, and a string to integer conversison function. For example, if the input to the first function is the integer 314, it should return the string "314" and if the input to the second function is the string "314" it should return the integer 314.

Hint: Build the result one digit at a time.

Solution: Let's consider the integer to string problem first. If the number to convert is a single digit, i.e., it is between 0 and 9, the result is easy to compute: it is the string consisting of the single character encoding that digit.

If the number has more than one digit, it is natural to perform the conversion digit-by-digit. The key insight is that for any positive integer x, the least significant digit in the decimal representation of x is $x \mod 10$, and the remaining digits are $x/10$. This approach computes the digits in reverse order, e.g., if we begin with 423, we get 3 and are left with 42 to convert. Then we get 2, and are left with 4 to convert. Finally, we get 4 and there are no digits to convert. The natural algorithm would be to prepend digits to the partial result. However, adding a digit to the beginning of a string is expensive, since all remaining digit have to be moved. A more time efficient approach is to add each computed digit to the end, and then reverse the computed sequence.

If x is negative, we record that, negate x, and then add a '-' before reversing. If x is 0, our code breaks out of the iteration without writing any digits, in which case we need to explicitly set a 0.

To convert from a string to an integer we recall the basic working of a positional number system. A base-10 number $d_2d_1d_0$ encodes the number $10^2 \times d_2 + 10^1 \times d_1 + d_0$. A brute-force algorithm then is to begin with the rightmost digit, and iteratively add $10^i \times d_i$ to a cumulative sum. The efficient way to compute 10^{i+1} is to use the existing value 10^i and multiply that by 10.

A more elegant solution is to begin from the leftmost digit and with each succeeding digit, multiply the partial result by 10 and add that digit. For example, to convert "314" to an integer, we

initial the partial result r to 0. In the first iteration, $r = 3$, in the second iteration $r = 3 \times 10 + 1 = 31$, and in the third iteration $r = 31 \times 10 + 4 = 314$, which is the final result.

Negative numbers are handled by recording the sign and negating the result.

```
def int_to_string(x):
    is_negative = False
    if x < 0:
        x, is_negative = -x, True

    s = []
    while True:
        s.append(chr(ord('0') + x % 10))
        x //= 10
        if x == 0:
            break

    # Adds the negative sign back if is_negative
    return ('-' if is_negative else '') + ''.join(reversed(s))


def string_to_int(s):
    return functools.reduce(
        lambda running_sum, c: running_sum * 10 + string.digits.index(c),
        s[s[0] == '-':], 0) * (-1 if s[0] == '-' else 1)
```

6.2 Base conversion

In the decimal number system, the position of a digit is used to signify the power of 10 that digit is to be multiplied with. For example, "314" denotes the number $3 \times 100 + 1 \times 10 + 4 \times 1$. The base b number system generalizes the decimal number system: the string "$a_{k-1}a_{k-2}\ldots a_1a_0$", where $0 \leq a_i < b$, denotes in base-b the integer $a_0 \times b^0 + a_1 \times b^1 + a_2 \times b^2 + \cdots + a_{k-1} \times b^{k-1}$.

Write a program that performs base conversion. The input is a string, an integer b_1, and another integer b_2. The string represents an integer in base b_1. The output should be the string representing the integer in base b_2. Assume $2 \leq b_1, b_2 \leq 16$. Use "A" to represent 10, "B" for 11, ..., and "F" for 15. (For example, if the string is "615", b_1 is 7 and b_2 is 13, then the result should be "1A7", since $6 \times 7^2 + 1 \times 7 + 5 = 1 \times 13^2 + 10 \times 13 + 7$.)

Hint: What base can you easily convert to and from?

Solution: A brute-force approach might be to convert the input to a unary representation, and then group the 1s as multiples of b_2, b_2^2, b_2^3, etc. For example, $(102)_3 = (11111111111)_1$. To convert to base 4, there are two groups of 4 and with three 1s remaining, so the result is $(23)_4$. This approach is hard to implement, and has terrible time and space complexity.

The insight to a good algorithm is the fact that all languages have an integer type, which supports arithmetical operations like multiply, add, divide, modulus, etc. These operations make the conversion much easier. Specifically, we can convert a string in base b_1 to integer type using a sequence of multiply and adds. Then we convert that integer type to a string in base b_2 using a sequence of modulus and division operations. For example, for the string is "615", $b_1 = 7$ and $b_2 = 13$, then the integer value, expressed in decimal, is 306. The least significant digit of the result is 306 mod 13 = 7, and we continue with 306/13 = 23. The next digit is 23 mod 13 = 10, which we

denote by 'A'. We continue with 23/13 = 1. Since 1 mod 13 = 1 and 1/13 = 0, the final digit is 1, and the overall result is "1A7". The design of the algorithm nicely illustrates the use of reduction.

Since the conversion algorithm is naturally expressed in terms of smaller similar subproblems, it is natural to implement it using recursion.

```
def convert_base(num_as_string, b1, b2):
    def construct_from_base(num_as_int, base):
        return ('' if num_as_int == 0 else
                construct_from_base(num_as_int // base, base) +
                string.hexdigits[num_as_int % base].upper())

    is_negative = num_as_string[0] == '-'
    num_as_int = functools.reduce(
        lambda x, c: x * b1 + string.hexdigits.index(c.lower()),
        num_as_string[is_negative:], 0)
    return ('-' if is_negative else '') + ('0' if num_as_int == 0 else
                                           construct_from_base(num_as_int, b2))
```

The time complexity is $O(n(1 + \log_{b_2} b_1))$, where n is the length of s. The reasoning is as follows. First, we perform n multiply-and-adds to get x from s. Then we perform $\log_{b_2} x$ multiply and adds to get the result. The value x is upper-bounded by b_1^n, and $\log_{b_2}(b_1^n) = n \log_{b_2} b_1$.

6.3 Compute the spreadsheet column encoding

Spreadsheets often use an alphabetical encoding of the successive columns. Specifically, columns are identified by "A", "B", "C", ..., "X", "Y", "Z", "AA", "AB", ..., "ZZ", "AAA", "AAB", ...

Implement a function that converts a spreadsheet column id to the corresponding integer, with "A" corresponding to 1. For example, you should return 4 for "D", 27 for "AA", 702 for "ZZ", etc. How would you test your code?

Hint: There are 26 characters in ["A", "Z"], and each can be mapped to an integer.

Solution: A brute-force approach could be to enumerate the column ids, stopping when the id equals the input. The logic for getting the successor of "Z", "AZ", etc. is slightly involved. The bigger issue is the time-complexity—it takes 26^6 steps to get to "ZZZZZZ". In general, the time complexity is $O(26^n)$, where n is the length of the string.

We can do better by taking larger jumps. Specifically, this problem is basically the problem of converting a string representing a base-26 number to the corresponding integer, except that "A" corresponds to 1 not 0. We can use the string to integer conversion approach given in Solution 6.1 on Page 68.

For example to convert "ZZ", we initialize result to 0. We add 26, multiply by 26, then add 26 again, i.e., the id is $26^2 + 26 = 702$.

Good test cases are around boundaries, e.g., "A", "B", "Y", "Z", "AA", "AB", "ZY", "ZZ", and some random strings, e.g., "M", "BZ", "CCC".

```
def ss_decode_col_id(col):
    return functools.reduce(
        lambda result, c: result * 26 + ord(c) - ord('A') + 1, col, 0)
```

The time complexity is $O(n)$.

Variant: Solve the same problem with "A" corresponding to 0.

Variant: Implement a function that converts an integer to the spreadsheet column id. For example, you should return "D" for 4, "AA" for 27, and "ZZ" for 702.

6.4 Replace and remove

Consider the following two rules that are to be applied to an array of characters.

- Replace each 'a' by two 'd's.
- Delete each entry containing a 'b'.

For example, applying these rules to the array $\langle a, c, d, b, b, c, a \rangle$ results in the array $\langle d, d, c, d, c, d, d \rangle$.

Write a program which takes as input an array of characters, and removes each 'b' and replaces each 'a' by two 'd's. Specifically, along with the array, you are provided an integer-valued size. Size denotes the number of entries of the array that the operation is to be applied to. You do not have to worry about preserving subsequent entries. For example, if the array is $\langle a, b, a, c, \_ \rangle$ and the size is 4, then you can return $\langle d, d, d, d, c \rangle$. You can assume there is enough space in the array to hold the final result.

Hint: Consider performing multiples passes on s.

Solution: Library array implementations often have methods for inserting into a specific location (all later entries are shifted right, and the array is resized) and deleting from a specific location (all later entries are shifted left, and the size of the array is decremented). If the input array had such methods, we could apply them—however, the time complexity would be $O(n^2)$, where n is the array's length. The reason is that each insertion and deletion from the array would have $O(n)$ time complexity.

This problem is trivial to solve in $O(n)$ time if we write result to a new array—we skip 'b's, replace 'a's by two 'd's, and copy over all other characters. However, this entails $O(n)$ additional space.

If there are no 'a's, we can implement the function without allocating additional space with one forward iteration by skipping 'b's and copying over the other characters.

If there are no 'b's, we can implement the function without additional space as follows. First, we compute the final length of the resulting string, which is the length of the array plus the number of 'a's. We can then write the result, character by character, starting from the last character, working our way backwards.

For example, suppose the array is $\langle a, c, a, a, \_, \_, \_ \rangle$, and the specified size is 4. Our algorithm updates the array to $\langle a, c, a, a, \_, \mathbf{d}, \mathbf{d} \rangle$. (Boldface denotes characters that are part of the final result.) The next update is $\langle a, c, a, \mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{d} \rangle$, followed by $\langle a, c, \mathbf{c}, \mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{d} \rangle$, and finally $\langle \mathbf{d}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{d} \rangle$.

We can combine these two approaches to get a complete algorithm. First, we delete 'b's and compute the final number of valid characters of the string, with a forward iteration through the string. Then we replace each 'a' by two 'd's, iterating backwards from the end of the resulting string. If there are more 'b's than 'a's, the number of valid entries will decrease, and if there are more 'a's than 'b's the number will increase. In the program below we return the number of valid entries in the final result.

```
def replace_and_remove(size, s):
    # Forward iteration: remove 'b's and count the number of 'a's.
```

```
    write_idx, a_count = 0, 0
    for i in range(size):
        if s[i] != 'b':
            s[write_idx] = s[i]
            write_idx += 1
        if s[i] == 'a':
            a_count += 1

    # Backward iteration: replace 'a's with 'dd's starting from the end.
    cur_idx = write_idx - 1
    write_idx += a_count - 1
    final_size = write_idx + 1
    while cur_idx >= 0:
        if s[cur_idx] == 'a':
            s[write_idx - 1:write_idx + 1] = 'dd'
            write_idx -= 2
        else:
            s[write_idx] = s[cur_idx]
            write_idx -= 1
        cur_idx -= 1
    return final_size
```

The forward and backward iterations each take $O(n)$ time, so the total time complexity is $O(n)$. No additional space is allocated.

Variant: You have an array C of characters. The characters may be letters, digits, blanks, and punctuation. The telex-encoding of the array C is an array T of characters in which letters, digits, and blanks appear as before, but punctuation marks are spelled out. For example, telex-encoding entails replacing the character "." by the string "DOT", the character "," by "COMMA", the character "?" by "QUESTION MARK", and the character "!" by "EXCLAMATION MARK". Design an algorithm to perform telex-encoding with $O(1)$ space.

Variant: Write a program which merges two sorted arrays of integers, A and B. Specifically, the final result should be a sorted array of length $m + n$, where n and m are the lengths of A and B, respectively. Use $O(1)$ additional storage—assume the result is stored in A, which has sufficient space. These arrays are C-style arrays, i.e., contiguous preallocated blocks of memory.

6.5 Test palindromicity

For the purpose of this problem, define a palindromic string to be a string which when all the nonalphanumeric are removed it reads the same front to back ignoring case. For example, "A man, a plan, a canal, Panama." and "Able was I, ere I saw Elba!" are palindromic, but "Ray a Ray" is not.

Implement a function which takes as input a string s and returns true if s is a palindromic string.

Hint: Use two indices.

Solution: The naive approach is to create a reversed version of s, and compare it with s, skipping nonalphanumeric characters. This requires additional space proportional to the length of s.

We do not need to create the reverse—rather, we can get the effect of the reverse of s by traversing s from right to left. Specifically, we use two indices to traverse the string, one forwards, the other backwards, skipping nonalphanumeric characters, performing case-insensitive comparison on the

alphanumeric characters. We return false as soon as there is a mismatch. If the indices cross, we have verified palindromicity.

```
def is_palindrome(s):
    # i moves forward, and j moves backward.
    i, j = 0, len(s) - 1
    while i < j:
        # i and j both skip non-alphanumeric characters.
        while not s[i].isalnum() and i < j:
            i += 1
        while not s[j].isalnum() and i < j:
            j -= 1
        if s[i].lower() != s[j].lower():
            return False
        i, j = i + 1, j - 1
    return True
```

We spend $O(1)$ per character, so the time complexity is $O(n)$, where n is the length of s.

6.6 Reverse all the words in a sentence

Given a string containing a set of words separated by whitespace, we would like to transform it to a string in which the words appear in the reverse order. For example, "Alice likes Bob" transforms to "Bob likes Alice". We do not need to keep the original string.

Implement a function for reversing the words in a string s.

Hint: It's difficult to solve this with one pass.

Solution: The code for computing the position for each character in the final result in a single pass is intricate.

However, for the special case where each word is a single character, the desired result is simply the reverse of s.

For the general case, reversing s gets the words to their correct relative positions. However, for words that are longer than one character, their letters appear in reverse order. This situation can be corrected by reversing the individual words.

For example, "ram is costly" reversed yields "yltsoc si mar". We obtain the final result by reversing each word to obtain "costly is ram".

```
# Assume s is a string encoded as bytearray.
def reverse_words(s):
    # First, reverse the whole string.
    s.reverse()

    def reverse_range(s, start, end):
        while start < end:
            s[start], s[end] = s[end], s[start]
            start, end = start + 1, end - 1

    start = 0
    while True:
        end = s.find(b' ', start)
        if end < 0:
            break
```

```
        # Reverses each word in the string.
        reverse_range(s, start, end - 1)
        start = end + 1
    # Reverses the last word.
    reverse_range(s, start, len(s) - 1)
```

Since we spend $O(1)$ per character, the time complexity is $O(n)$, where n is the length of s. The computation in place, i.e., the additional space complexity is $O(1)$.

6.7 Compute all mnemonics for a phone number

Each digit, apart from 0 and 1, in a phone keypad corresponds to one of three or four letters of the alphabet, as shown in Figure 6.1. Since words are easier to remember than numbers, it is natural to ask if a 7 or 10-digit phone number can be represented by a word. For example, "2276696" corresponds to "ACRONYM" as well as "ABPOMZN".

Figure 6.1: Phone keypad.

Write a program which takes as input a phone number, specified as a string of digits, and returns all possible character sequences that correspond to the phone number. The cell phone keypad is specified by a mapping that takes a digit and returns the corresponding set of characters. The character sequences do not have to be legal words or phrases.

Hint: Use recursion.

Solution: For a 7 digit phone number, the brute-force approach is to form 7 ranges of characters, one for each digit. For example, if the number is "2276696" then the ranges are 'A'–'C', 'A'–'C', 'P'–'S', 'M'–'O', 'M'–'O', 'W'–'Z', and 'M'–'O'. We use 7 nested for-loops where the iteration variables correspond to the 7 ranges to enumerate all possible mnemonics. The drawbacks of such an approach are its repetitiveness in code and its inflexibility.

As a general rule, any such enumeration is best computed using recursion. The execution path is very similar to that of the brute-force approach, but the compiler handles the looping.

```
# The mapping from digit to corresponding characters.
MAPPING = ('0', '1', 'ABC', 'DEF', 'GHI', 'JKL', 'MNO', 'PQRS', 'TUV', 'WXYZ')


def phone_mnemonic(phone_number):
    def phone_mnemonic_helper(digit):
        if digit == len(phone_number):
```

```
            # All digits are processed, so add partial_mnemonic to mnemonics.
            # (We add a copy since subsequent calls modify partial_mnemonic.)
            mnemonics.append(''.join(partial_mnemonic))
        else:
            # Try all possible characters for this digit.
            for c in MAPPING[int(phone_number[digit])]:
                partial_mnemonic[digit] = c
                phone_mnemonic_helper(digit + 1)

    mnemonics, partial_mnemonic = [], [0] * len(phone_number)
    phone_mnemonic_helper(0)
    return mnemonics
```

Since there are no more than 4 possible characters for each digit, the number of recursive calls, $T(n)$, satisfies $T(n) \leq 4T(n-1)$, where n is the number of digits in the number. This solves to $T(n) = O(4^n)$. For the function calls that entail recursion, the time spent within the function, not including the recursive calls, is $O(1)$. Each base case entails making a copy of a string and adding it to the result. Since each such string has length n, each base case takes time $O(n)$. Therefore, the time complexity is $O(4^n n)$.

Variant: Solve the same problem without using recursion.

6.8 The look-and-say problem

The look-and-say sequence starts with 1. Subsequent numbers are derived by describing the previous number in terms of consecutive digits. Specifically, to generate an entry of the sequence from the previous entry, read off the digits of the previous entry, counting the number of digits in groups of the same digit. For example, 1; one 1; two 1s; one 2 then one 1; one 1, then one 2, then two 1s; three 1s, then two 2s, then one 1. The first eight numbers in the look-and-say sequence are $\langle 1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211 \rangle$.

Write a program that takes as input an integer n and returns the nth integer in the look-and-say sequence. Return the result as a string.

Hint: You need to return the result as a string.

Solution: We compute the nth number by iteratively applying this rule $n-1$ times. Since we are counting digits, it is natural to use strings to represent the integers in the sequence. Specifically, going from the ith number to the $(i+1)$th number entails scanning the digits from most significant to least significant, counting the number of consecutive equal digits, and writing these counts.

```
def look_and_say(n):
    def next_number(s):
        result, i = [], 0
        while i < len(s):
            count = 1
            while i + 1 < len(s) and s[i] == s[i + 1]:
                i += 1
                count += 1
            result.append(str(count) + s[i])
            i += 1
        return ''.join(result)
```

```
    s = '1'
    for _ in range(1, n):
        s = next_number(s)
    return s


# Pythonic solution uses the power of itertools.groupby().
def look_and_say_pythonic(n):
    s = '1'
    for _ in range(n - 1):
        s = ''.join(
            str(len(list(group))) + key for key, group in itertools.groupby(s))
    return s
```

The precise time complexity is a function of the lengths of the terms, which is extremely hard to analyze. Each successive number can have at most twice as many digits as the previous number—this happens when all digits are different. This means the maximum length number has length no more than 2^n. Since there are n iterations and the work in each iteration is proportional to the length of the number computed in the iteration, a simple bound on the time complexity is $O(n2^n)$.

6.9 Convert from Roman to decimal

The Roman numeral representation of positive integers uses the symbols I, V, X, L, C, D, M. Each symbol represents a value, with I being 1, V being 5, X being 10, L being 50, C being 100, D being 500, and M being 1000.

In this problem we give simplified rules for representing numbers in this system. Specifically, define a string over the Roman number symbols to be a valid Roman number string if symbols appear in nonincreasing order, with the following exceptions allowed:

- I can immediately precede V and X.
- X can immediately precede L and C.
- C can immediately precede D and M.

Back-to-back exceptions are not allowed, e.g., IXC is invalid, as is CDM.

A valid complex Roman number string represents the integer which is the sum of the symbols that do not correspond to exceptions; for the exceptions, add the difference of the larger symbol and the smaller symbol.

For example, the strings "XXXXXIIIIIIIII", "LVIIII" and "LIX" are valid Roman number strings representing 59. The shortest valid complex Roman number string corresponding to the integer 59 is "LIX".

Write a program which takes as input a valid Roman number string s and returns the integer it corresponds to.

Hint: Start by solving the problem assuming no exception cases.

Solution: The brute-force approach is to scan s from left to right, adding the value for the corresponding symbol unless the symbol subsequent to the one being considered has a higher value, in which case the pair is one of the six exception cases and the value of the pair is added.

A slightly easier-to-code solution is to start from the right, and if the symbol after the current one is greater than it, we subtract the current symbol. The code below performs the right-to-left

iteration. It does not check that when a smaller symbol appears to the left of a larger one that it is one of the six allowed exceptions, so it will, for example, return 99 for "IC".

```
def roman_to_integer(s):
    T = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}

    return functools.reduce(
        lambda val, i: val + (-T[s[i]] if T[s[i]] < T[s[i + 1]] else T[s[i]]),
        reversed(range(len(s) - 1)), T[s[-1]])
```

Each character of s is processed in $O(1)$ time, yielding an $O(n)$ overall time complexity, where n is the length of s.

Variant: Write a program that takes as input a string of Roman number symbols and checks whether that string is valid.

Variant: Write a program that takes as input a positive integer n and returns a shortest valid simple Roman number string representing n.

6.10 Compute all valid IP addresses

A decimal string is a string consisting of digits between 0 and 9. Internet Protocol (IP) addresses can be written as four decimal strings separated by periods, e.g., 192.168.1.201. A careless programmer mangles a string representing an IP address in such a way that all the periods vanish.

Write a program that determines where to add periods to a decimal string so that the resulting string is a valid IP address. There may be more than one valid IP address corresponding to a string, in which case you should print all possibilities.

For example, if the mangled string is "19216811" then two corresponding IP addresses are 192.168.1.1 and 19.216.81.1. (There are seven other possible IP addresses for this string.)

Hint: Use nested loops.

Solution: There are three periods in a valid IP address, so we can enumerate all possible placements of these periods, and check whether all four corresponding substrings are between 0 and 255. We can reduce the number of placements considered by spacing the periods 1 to 3 characters apart. We can also prune by stopping as soon as a substring is not valid.

For example, if the string is "19216811", we could put the first period after "1", "19", and "192". If the first part is "1", the second part could be "9", "92", and "921". Of these, "921" is illegal so we do not continue with it.

```
def get_valid_ip_address(s):
    def is_valid_part(s):
        # '00', '000', '01', etc. are not valid, but '0' is valid.
        return len(s) == 1 or (s[0] != '0' and int(s) <= 255)

    result, parts = [], [None] * 4
    for i in range(1, min(4, len(s))):
        parts[0] = s[:i]
        if is_valid_part(parts[0]):
            for j in range(1, min(len(s) - i, 4)):
                parts[1] = s[i:i + j]
                if is_valid_part(parts[1]):
```

```
                for k in range(1, min(len(s) - i - j, 4)):
                    parts[2], parts[3] = s[i + j:i + j + k], s[i + j + k:]
                    if is_valid_part(parts[2]) and is_valid_part(parts[3]):
                        result.append('.'.join(parts))
    return result
```

The total number of IP addresses is a constant (2^{32}), implying an $O(1)$ time complexity for the above algorithm.

Variant: Solve the analogous problem when the number of periods is a parameter k and the string length is unbounded.

6.11 Write a string sinusoidally

We illustrate what it means to write a string in sinusoidal fashion by means of an example. The string "Hello␣World!" written in sinusoidal fashion is

```
      e       ␣       l
H     l   o   W   r   d
        l       o       !
```

(Here ␣ denotes a blank.)

Define the snakestring of s to be the left-right top-to-bottom sequence in which characters appear when s is written in sinusoidal fashion. For example, the snakestring string for "Hello␣World!" is "e␣lHloWrdlo!".

Write a program which takes as input a string s and returns the snakestring of s.

Hint: Try concrete examples, and look for periodicity.

Solution: The brute-force approach is to populate a $3 \times n$ 2D array of characters, initialized to null entries. We then write the string in sinusoidal manner in this array. Finally, we read out the non-null characters in row-major manner.

However, observe that the result begins with the characters $s[1], s[5], s[9], \ldots$, followed by $s[0], s[2], s[4], \ldots$, and then $s[3], s[7], s[11], \ldots$. Therefore, we can create the snakestring directly, with three iterations through s.

```
def snake_string(s):
    result = []
    # Outputs the first row, i.e., s[1], s[5], s[9], ...
    for i in range(1, len(s), 4):
        result.append(s[i])
    # Outputs the second row, i.e., s[0], s[2], s[4], ...
    for i in range(0, len(s), 2):
        result.append(s[i])
    # Outputs the third row, i.e., s[3], s[7], s[11], ...
    for i in range(3, len(s), 4):
        result.append(s[i])
    return ''.join(result)


# Python solution uses slicing with right steps.
def snake_string_pythonic(s):
    return s[1::4] + s[::2] + s[3::4]
```

Let n be the length of s. Each of the three iterations takes $O(n)$ time, implying an $O(n)$ time complexity.

6.12 Implement run-length encoding

Run-length encoding (RLE) compression offers a fast way to do efficient on-the-fly compression and decompression of strings. The idea is simple—encode successive repeated characters by the repetition count and the character. For example, the RLE of "aaaabcccaa" is "4a1b3c2a". The decoding of "3e4f2e" returns "eeefffee".

Implement run-length encoding and decoding functions. Assume the string to be encoded consists of letters of the alphabet, with no digits, and the string to be decoded is a valid encoding.

Hint: This is similar to converting between binary and string representations.

Solution: First we consider the decoding function. Every encoded string is a repetition of a string of digits followed by a single character. The string of digits is the decimal representation of a positive integer. To generate the decoded string, we need to convert this sequence of digits into its integer equivalent and then write the character that many times. We do this for each character.

The encoding function requires an integer (the repetition count) to string conversion.

```
def decoding(s):
    count, result = 0, []
    for c in s:
        if c.isdigit():
            count = count * 10 + int(c)
        else:  # c is a letter of alphabet.
            result.append(c * count)  # Appends count copies of c to result.
            count = 0
    return ''.join(result)


def encoding(s):
    result, count = [], 1
    for i in range(1, len(s) + 1):
        if i == len(s) or s[i] != s[i - 1]:
            # Found new character so write the count of previous character.
            result.append(str(count) + s[i - 1])
            count = 1
        else:  # s[i] == s[i - 1].
            count += 1
    return ''.join(result)
```

The time complexity is $O(n)$, where n is the length of the string.

6.13 Find the first occurrence of a substring

A good string search algorithm is fundamental to the performance of many applications. Several clever algorithms have been proposed for string search, each with its own trade-offs. As a result, there is no single perfect answer. If someone asks you this question in an interview, the best way to approach this problem would be to work through one good algorithm in detail and discuss at a high level other algorithms.

Given two strings s (the "search string") and t (the "text"), find the first occurrence of s in t.

Hint: Form a signature from a string.

Solution: The brute-force algorithm uses two nested loops, the first iterates through t, the second tests if s occurs starting at the current index in t. The worst-case complexity is high. If t consists of n 'a's and s is $n/2$ 'a's followed by a 'b', it will perform $n/2$ unsuccessful string compares, each of which entails $n/2 + 1$ character compares, so the brute-force algorithm's time complexity is $O(n^2)$.

Intuitively, the brute-force algorithm is slow because it advances through t one character at a time, and potentially does $O(m)$ computation with each advance, where m is the length of s.

There are three linear time string matching algorithms: KMP, Boyer-Moore, and Rabin-Karp. Of these, Rabin-Karp is by far the simplest to understand and implement.

The Rabin-Karp algorithm is very similar to the brute-force algorithm, but it does not require the second loop. Instead it uses the concept of a "fingerprint". Specifically, let m be the length of s. It computes hash codes of each substring whose length is m—these are the fingerprints. The key to efficiency is using an incremental hash function, such as a function with the property that the hash code of a string is an additive function of each individual character. (Such a hash function is sometimes referred to as a rolling hash.) For such a function, getting the hash code of a sliding window of characters is very fast for each shift.

For example, let the strings consist of letters from $\{A, C, G, T\}$. Suppose t is "GACGCCA" and s is "CGC". Define the code for "A" to be 0, the code for "C" to be 1, etc. Let the hash function be the decimal number formed by the integer codes for the letters. The hash code of s is 121. The hash code of the first three characters of t, "GAC", is 201, so s cannot be the first three characters of t. Continuing, the next substring of t is "ACG", whose hash code can be computed from 201 by subtracting 200, then multiplying by 10, and finally adding 2. This yields 12, so there no match yet. We then reach "CGC" whose hash code, 121, is derived in a similar manner. We are not done yet—there may be a collision. We check explicitly if the substring matches s, which in this case it does.

For the Rabin-Karp algorithm to run in linear time, we need a good hash function, to reduce the likelihood of collisions, which entail potentially time consuming string equality checks.

```
def rabin_karp(t, s):
    if len(s) > len(t):
        return -1  # s is not a substring of t.

    BASE = 26
    # Hash codes for the substring of t and s.
    t_hash = functools.reduce(lambda h, c: h * BASE + ord(c), t[:len(s)], 0)
    s_hash = functools.reduce(lambda h, c: h * BASE + ord(c), s, 0)
    power_s = BASE**max(len(s) - 1, 0)  # BASE^|s-1|.

    for i in range(len(s), len(t)):
        # Checks the two substrings are actually equal or not, to protect
        # against hash collision.
        if t_hash == s_hash and t[i - len(s):i] == s:
            return i - len(s)  # Found a match.

        # Uses rolling hash to compute the hash code.
        t_hash -= ord(t[i - len(s)]) * power_s
        t_hash = t_hash * BASE + ord(t[i])
```

```
    # Tries to match s and t[-len(s):].
    if t_hash == s_hash and t[-len(s):] == s:
        return len(t) - len(s)
    return -1  # s is not a substring of t.
```

For a good hash function, the time complexity is $O(m+n)$, independent of the inputs s and t, where m is the length of s and n is the length of t.

CHAPTER

7 Linked Lists

The S-expressions are formed according to the following recursive rules.

1. *The atomic symbols p_1, p_2, etc., are S-expressions.*
2. *A null expression $\wedge$ is also admitted.*
3. *If e is an S-expression so is (e).*
4. *If e_1 and e_2 are S-expressions so is (e_1, e_2).*

— *"Recursive Functions Of Symbolic Expressions,"* J. McCARTHY, 1959

A list implements an ordered collection of values, which may include repetitions. Sepcifically, a *singly linked list* is a data structure that contains a sequence of nodes such that each node contains an object and a reference to the next node in the list. The first node is referred to as the *head* and the last node is referred to as the *tail*; the tail's next field is null. The structure of a singly linked list is given in Figure 7.1. There are many variants of linked lists, e.g., in a *doubly linked list*, each node has a link to its predecessor; similarly, a sentinel node or a self-loop can be used instead of null to mark the end of the list. The structure of a doubly linked list is given in Figure 7.2.

A list is similar to an array in that it contains objects in a linear order. The key differences are that inserting and deleting elements in a list has time complexity $O(1)$. On the other hand, obtaining the kth element in a list is expensive, having $O(n)$ time complexity. Lists are usually building blocks of more complex data structures. However, as we will see in this chapter, they can be the subject of tricky problems in their own right.

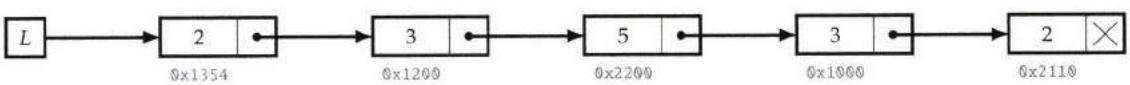


Figure 7.1: Example of a singly linked list. The number in hex below a node indicates the memory address of that node.

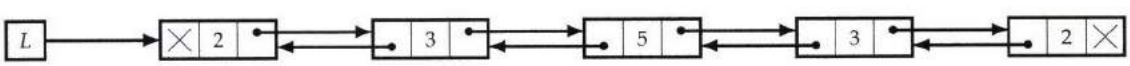


Figure 7.2: Example of a doubly linked list.

For all problems in this chapter, unless otherwise stated, each node has two entries—a data field, and a next field, which points to the next node in the list, with the next field of the last node being null. Its prototype is as follows:

```
class ListNode:
    def __init__(self, data=0, next_node=None):
        self.data = data
        self.next = next_node
```

Linked lists boot camp

There are two types of list-related problems—those where you have to implement your own list, and those where you have to exploit the standard list library. We will review both these aspects here, starting with implementation, then moving on to list libraries.

Implementing a basic list API—search, insert, delete—for singly linked lists is an excellent way to become comfortable with lists.

Search for a key:

```
def search_list(L, key):
    while L and L.data != key:
        L = L.next
    # If key was not present in the list, L will have become null.
    return L
```

Insert a new node after a specified node:

```
# Insert new_node after node.
def insert_after(node, new_node):
    new_node.next = node.next
    node.next = new_node
```

Delete a node:

```
# Delete the node past this one. Assume node is not a tail.
def delete_after(node):
    node.next = node.next.next
```

Insert and delete are local operations and have $O(1)$ time complexity. Search requires traversing the entire list, e.g., if the key is at the last node or is absent, so its time complexity is $O(n)$, where n is the number of nodes.

List problems often have a simple brute-force solution that uses $O(n)$ space, but a subtler solution that uses the **existing list nodes** to reduce space complexity to $O(1)$.
Very often, a problem on lists is conceptually simple, and is more about **cleanly coding what's specified**, rather than designing an algorithm.
Consider using a **dummy head** (sometimes referred to as a sentinel) to avoid having to check for empty lists. This simplifies code, and makes bugs less likely.
It's easy to forget to **update next** (and previous for double linked list) for the head and tail.
Algorithms operating on singly linked lists often benefit from using **two iterators**, one ahead of the other, or one advancing quicker than the other.

Table 7.1: Top Tips for Linked Lists

Know your linked list libraries

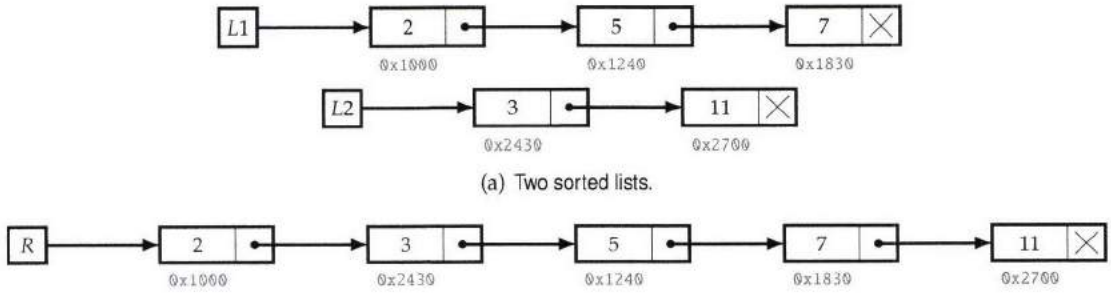
We now review the standard linked list library, with the reminder that many interview problems that are directly concerned with lists require you to write your own list class.

Under the hood, the Python `list` type is typically implemented as a dynamically resized array, and the key methods on it are described on Page 38. This chapter is concerned specifically with

linked lists, which are not a standard type in Python. We define our own singly and doubly linked list types. Some of the key methods on these lists include returning the head/tail, adding an element at the head/tail, returning the value stored at the head/tail, and deleting the head, tail, or arbitrary node in the list.

7.1 Merge two sorted lists

Consider two singly linked lists in which each node holds a number. Assume the lists are sorted, i.e., numbers in the lists appear in ascending order within each list. The *merge* of the two lists is a list consisting of the nodes of the two lists in which numbers appear in ascending order. Merge is illustrated in Figure 7.3.



(a) Two sorted lists.

(b) The merge of the lists in (a).

Figure 7.3: Merging sorted lists.

Write a program that takes two lists, assumed to be sorted, and returns their merge. The only field your program can change in a node is its next field.

Hint: Two sorted arrays can be merged using two indices. For lists, take care when one iterator reaches the end.

Solution: A naive approach is to append the two lists together and sort the resulting list. The drawback of this approach is that it does not use the fact that the initial lists are sorted. The time complexity is that of sorting, which is $O((n+m)\log(n+m))$, where n and m are the lengths of each of the two input lists.

A better approach, in terms of time complexity, is to traverse the two lists, always choosing the node containing the smaller key to continue traversing from.

```
def merge_two_sorted_lists(L1, L2):
    # Creates a placeholder for the result.
    dummy_head = tail = ListNode()

    while L1 and L2:
        if L1.data < L2.data:
            tail.next, L1 = L1, L1.next
        else:
            tail.next, L2 = L2, L2.next
        tail = tail.next

    # Appends the remaining nodes of L1 or L2
    tail.next = L1 or L2
    return dummy_head.next
```

The worst-case, from a runtime perspective, corresponds to the case when the lists are of comparable length, so the time complexity is $O(n + m)$. (In the best-case, one list is much shorter than the other and all its entries appear at the beginning of the merged list.) Since we reuse the existing nodes, the space complexity is $O(1)$.

Variant: Solve the same problem when the lists are doubly linked.

7.2 Reverse a single sublist

This problem is concerned with reversing a sublist within a list. See Figure 7.4 for an example of sublist reversal.

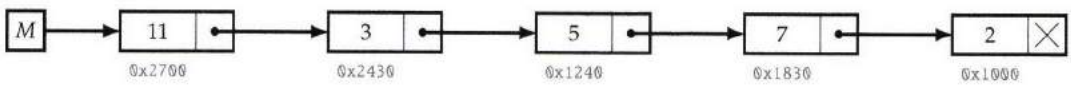


Figure 7.4: The result of reversing the sublist consisting of the second to the fourth nodes, inclusive, in the list in Figure 7.5 on the following page.

Write a program which takes a singly linked list L and two integers s and f as arguments, and reverses the order of the nodes from the sth node to fth node, inclusive. The numbering begins at 1, i.e., the head node is the first node. Do not allocate additional nodes.

Hint: Focus on the successor fields which have to be updated.

Solution: The direct approach is to extract the sublist, reverse it, and splice it back in. The drawback for this approach is that it requires two passes over the sublist.

The update can be performed with a single pass by combining the identification of the sublist with its reversal. We identify the start of sublist by using an iteration to get the sth node and its predecessor. Once we reach the sth node, we start the reversing process and keep counting. When we reach the fth node, we stop the reversion process and link the reverted section with the unreverted sections.

```
def reverse_sublist(L, start, finish):
    dummy_head = sublist_head = ListNode(0, L)
    for _ in range(1, start):
        sublist_head = sublist_head.next

    # Reverses sublist.
    sublist_iter = sublist_head.next
    for _ in range(finish - start):
        temp = sublist_iter.next
        sublist_iter.next, temp.next, sublist_head.next = (temp.next,
                                                           sublist_head.next,
                                                           temp)
    return dummy_head.next
```

The time complexity is dominated by the search for the fth node, i.e., $O(f)$.

Variant: Write a function that reverses a singly linked list. The function should use no more than constant storage beyond that needed for the list itself. The desired transformation is illustrated in Figure 7.5 on the next page.

Variant: Write a program which takes as input a singly linked list L and a nonnegative integer k, and reverses the list k nodes at a time. If the number of nodes n in the list is not a multiple of k, leave the last n mod k nodes unchanged. Do not change the data stored within a node.

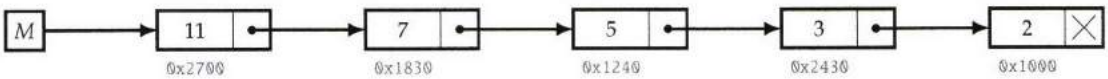


Figure 7.5: The reversed list for the list in Figure 7.3(b) on Page 84. Note that no new nodes have been allocated.

7.3 Test for cyclicity

Although a linked list is supposed to be a sequence of nodes ending in null, it is possible to create a cycle in a linked list by making the next field of an element reference to one of the earlier nodes.

Write a program that takes the head of a singly linked list and returns null if there does not exist a cycle, and the node at the start of the cycle, if a cycle is present. (You do not know the length of the list in advance.)

Hint: Consider using two iterators, one fast and one slow.

Solution: This problem has several solutions. If space is not an issue, the simplest approach is to explore nodes via the next field starting from the head and storing visited nodes in a hash table—a cycle exists if and only if we visit a node already in the hash table. If no cycle exists, the search ends at the tail (often represented by having the next field set to null). This solution requires $O(n)$ space, where n is the number of nodes in the list.

A brute-force approach that does not use additional storage and does not modify the list is to traverse the list in two loops—the outer loop traverses the nodes one-by-one, and the inner loop starts from the head, and traverses as many nodes as the outer loop has gone through so far. If the node being visited by the outer loop is visited twice, a loop has been detected. (If the outer loop encounters the end of the list, no cycle exists.) This approach has $O(n^2)$ time complexity.

This idea can be made to work in linear time—use a slow iterator and a fast iterator to traverse the list. In each iteration, advance the slow iterator by one and the fast iterator by two. The list has a cycle if and only if the two iterators meet. The reasoning is as follows: if the fast iterator jumps over the slow iterator, the slow iterator will equal the fast iterator in the next step.

Now, assuming that we have detected a cycle using the above method, we can find the start of the cycle, by first calculating the cycle length C. Once we know there is a cycle, and we have a node on it, it is trivial to compute the cycle length. To find the first node on the cycle, we use two iterators, one of which is C ahead of the other. We advance them in tandem, and when they meet, that node must be the first node on the cycle.

The code to do this traversal is quite simple:

```
def has_cycle(head):
    def cycle_len(end):
        start, step = end, 0
        while True:
            step += 1
            start = start.next
            if start is end:
                return step
```

```
    fast = slow = head
    while fast and fast.next and fast.next.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast:
            # Finds the start of the cycle.
            cycle_len_advanced_iter = head
            for _ in range(cycle_len(slow)):
                cycle_len_advanced_iter = cycle_len_advanced_iter.next

            it = head
            # Both iterators advance in tandem.
            while it is not cycle_len_advanced_iter:
                it = it.next
                cycle_len_advanced_iter = cycle_len_advanced_iter.next
            return it  # iter is the start of cycle.
    return None  # No cycle.
```

Let F be the number of nodes to the start of the cycle, C the number of nodes on the cycle, and n the total number of nodes. Then the time complexity is $O(F) + O(C) = O(n)$—$O(F)$ for both pointers to reach the cycle, and $O(C)$ for them to overlap once the slower one enters the cycle.

Variant: The following program purports to compute the beginning of the cycle without determining the length of the cycle; it has the benefit of being more succinct than the code listed above. Is the program correct?

```
def has_cycle(head):
    fast = slow = head
    while fast and fast.next and fast.next.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast:  # There is a cycle.
            # Tries to find the start of the cycle.
            slow = head
            # Both pointers advance at the same time.
            while slow is not fast:
                slow, fast = slow.next, fast.next
            return slow  # slow is the start of cycle.
    return None  # No cycle.
```

7.4 Test for overlapping lists—lists are cycle-free

Given two singly linked lists there may be list nodes that are common to both. (This may not be a bug—it may be desirable from the perspective of reducing memory footprint, as in the flyweight pattern, or maintaining a canonical form.) For example, the lists in Figure 7.6 on the following page overlap at Node *I*.

Write a program that takes two cycle-free singly linked lists, and determines if there exists a node that is common to both lists.

Hint: Solve the simple cases first.

Solution: A brute-force approach is to store one list's nodes in a hash table, and then iterate through the nodes of the other, testing each for presence in the hash table. This takes $O(n)$ time and $O(n)$ space, where n is the total number of nodes.

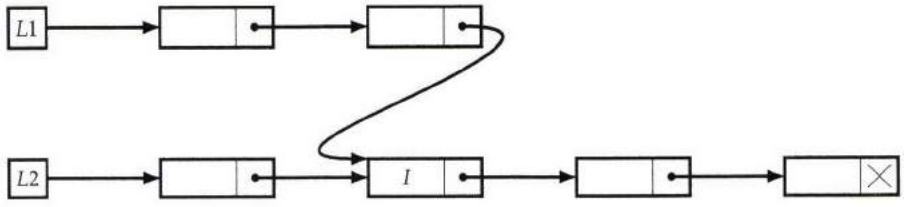


Figure 7.6: Example of overlapping lists.

We can avoid the extra space by using two nested loops, one iterating through the first list, and the other to search the second for the node being processed in the first list. However, the time complexity is $O(n^2)$.

The lists overlap if and only if both have the same tail node: once the lists converge at a node, they cannot diverge at a later node. Therefore, checking for overlap amounts to finding the tail nodes for each list.

To find the first overlapping node, we first compute the length of each list. The first overlapping node is determined by advancing through the longer list by the difference in lengths, and then advancing through both lists in tandem, stopping at the first common node. If we reach the end of a list without finding a common node, the lists do not overlap.

```
def overlapping_no_cycle_lists(L1, L2):
    def length(L):
        length = 0
        while L:
            length += 1
            L = L.next
        return length

    L1_len, L2_len = length(L1), length(L2)
    if L1_len > L2_len:
        L1, L2 = L2, L1  # L2 is the longer list
    # Advances the longer list to get equal length lists.
    for _ in range(abs(L1_len - L2_len)):
        L2 = L2.next

    while L1 and L2 and L1 is not L2:
        L1, L2 = L1.next, L2.next
    return L1  # None implies there is no overlap between L1 and L2.
```

The time complexity is $O(n)$ and the space complexity is $O(1)$.

7.5 Test for overlapping lists—lists may have cycles

Solve Problem 7.4 on the previous page for the case where the lists may each or both have a cycle. If such a node exists, return a node that appears first when traversing the lists. This node may not be unique—if one node ends in a cycle, the first cycle node encountered when traversing it may be different from the first cycle node encountered when traversing the second list, even though the cycle is the same. In such cases, you may return either of the two nodes.

For example, Figure 7.7 on the facing page shows an example of lists which overlap and have cycles. For this example, both A and B are acceptable answers.

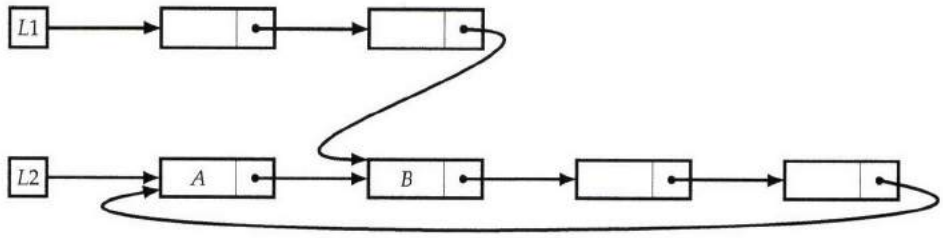


Figure 7.7: Overlapping lists.

Hint: Use case analysis. What if both lists have cycles? What if they end in a common cycle? What if one list has cycle and the other does not?

Solution: This problem is easy to solve using $O(n)$ time and space complexity, where n is the total number of nodes, using the hash table approach in Solution 7.4 on Page 87.

We can improve space complexity by studying different cases. The easiest case is when neither list is cyclic, which we can determine using Solution 7.3 on Page 86. In this case, we can check overlap using the technique in Solution 7.4 on Page 87.

If one list is cyclic, and the other is not, they cannot overlap, so we are done.

This leaves us with the case that both lists are cyclic. In this case, if they overlap, the cycles must be identical.

There are two subcases: the paths to the cycle merge before the cycle, in which case there is a unique first node that is common, or the paths reach the cycle at different nodes on the cycle. For the first case, we can use the approach of Solution 7.4 on Page 87. For the second case, we use the technique in Solution 7.3 on Page 86.

```
def overlapping_lists(L1, L2):
    # Store the start of cycle if any.
    root1, root2 = has_cycle(L1), has_cycle(L2)

    if not root1 and not root2:
        # Both lists don't have cycles.
        return overlapping_no_cycle_lists(L1, L2)
    elif (root1 and not root2) or (not root1 and root2):
        # One list has cycle, one list has no cycle.
        return None
    # Both lists have cycles.
    temp = root2
    while True:
        temp = temp.next
        if temp is root1 or temp is root2:
            break

    # L1 and L2 do not end in the same cycle.
    if temp is not root1:
        return None  # Cycles are disjoint.

    # Calculates the distance between a and b.
    def distance(a, b):
        dis = 0
        while a is not b:
            a = a.next
```

```
            dis += 1
        return dis

    # L1 and L2 end in the same cycle, locate the overlapping node if they
    # first overlap before cycle starts.
    stem1_length, stem2_length = distance(L1, root1), distance(L2, root2)
    if stem1_length > stem2_length:
        L2, L1 = L1, L2
        root1, root2 = root2, root1
    for _ in range(abs(stem1_length - stem2_length)):
        L2 = L2.next
    while L1 is not L2 and L1 is not root1 and L2 is not root2:
        L1, L2 = L1.next, L2.next

    # If L1 == L2 before reaching root1, it means the overlap first occurs
    # before the cycle starts; otherwise, the first overlapping node is not
    # unique, we can return any node on the cycle.
    return L1 if L1 is L2 else root1
```

The algorithm has time complexity $O(n + m)$, where n and m are the lengths of the input lists, and space complexity $O(1)$.

7.6 Delete a node from a singly linked list

Given a node in a singly linked list, deleting it in $O(1)$ time appears impossible because its predecessor's next field has to be updated. Surprisingly, it can be done with one small caveat—the node to delete cannot be the last one in the list and it is easy to copy the value part of a node.

Write a program which deletes a node in a singly linked list. The input node is guaranteed not to be the tail node.

Hint: Instead of deleting the node, can you delete its successor and still achieve the desired configuration?

Solution: Given the pointer to a node, it is impossible to delete it from the list without modifying its predecessor's next pointer and the only way to get to the predecessor is to traverse the list from head, which requires $O(n)$ time, where n is the number of nodes in the list.

Given a node, it is easy to delete its successor, since this just requires updating the next pointer of the current node. If we copy the value part of the next node to the current node, and then delete the next node, we have effectively deleted the current node. The time complexity is $O(1)$.

```
# Assumes node_to_delete is not tail.
def deletion_from_list(node_to_delete):
    node_to_delete.data = node_to_delete.next.data
    node_to_delete.next = node_to_delete.next.next
```

7.7 Remove the kth last element from a list

Without knowing the length of a linked list, it is not trivial to delete the kth last element in a singly linked list.

Given a singly linked list and an integer k, write a program to remove the kth last element from the list. Your algorithm cannot use more than a few words of storage, regardless of the length of the list. In particular, you cannot assume that it is possible to record the length of the list.

Hint: If you know the length of the list, can you find the kth last node using two iterators?

Solution: A brute-force approach is to compute the length with one pass, and then use that to determine which node to delete in a second pass. A drawback of this approach is that it entails two passes over the data, which is slow, e.g., if traversing the list entails disc accesses.

We use two iterators to traverse the list. The first iterator is advanced by k steps, and then the two iterators advance in tandem. When the first iterator reaches the tail, the second iterator is at the $(k+1)$th last node, and we can remove the kth node.

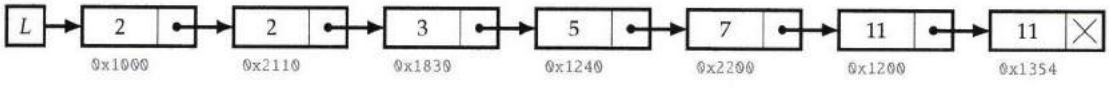
```
# Assumes L has at least k nodes, deletes the k-th last node in L.
def remove_kth_last(L, k):
    dummy_head = ListNode(0, L)
    first = dummy_head.next
    for _ in range(k):
        first = first.next

    second = dummy_head
    while first:
        first, second = first.next, second.next
    # second points to the (k + 1)-th last node, deletes its successor.
    second.next = second.next.next
    return dummy_head.next
```

The time complexity is that of list traversal, i.e., $O(n)$, where n is the length of the list. The space complexity is $O(1)$, since there are only two iterators.

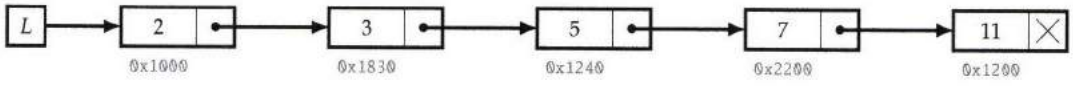
Compared to the brute-force approach, if k is small enough that we can keep the set of nodes between the two iterators in memory, but the list is too big to fit in memory, the two-iterator approach halves the number of disc accesses.

7.8 Remove duplicates from a sorted list

This problem is concerned with removing duplicates from a sorted list of integers. See Figure 7.8 for an example.



(a) List before removing duplicates.



(b) The list in (a) after removing duplicates.

Figure 7.8: Example of duplicate removal.

Write a program that takes as input a singly linked list of integers in sorted order, and removes duplicates from it. The list should be sorted.

Hint: Focus on the successor fields which have to be updated.

Solution: A brute-force algorithm is to create a new list, using a hash table to test if a value has already been added to the new list. Alternatively, we could search in the new list itself to see if

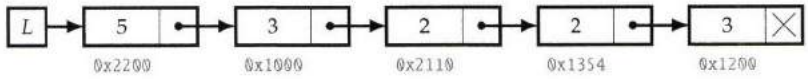


Figure 7.9: The result of applying a right cyclic shift by 3 to the list in Figure 7.1 on Page 82. Note that no new nodes have been allocated.

the candidate value already is present. If the length of the list is n, the first approach requires $O(n)$ additional space for the hash table, and the second requires $O(n^2)$ time to perform the lookups. Both allocate n nodes for the new list.

A better approach is to exploit the sorted nature of the list. As we traverse the list, we remove all successive nodes with the same value as the current node.

```
def remove_duplicates(L):
    it = L
    while it:
        # Uses next_distinct to find the next distinct value.
        next_distinct = it.next
        while next_distinct and next_distinct.data == it.data:
            next_distinct = next_distinct.next
        it.next = next_distinct
        it = next_distinct
    return L
```

Determining the time complexity requires a little amortized analysis. A single node may take more than $O(1)$ time to process if there are many successive nodes with the same value. A clearer justification for the time complexity is that each link is traversed once, so the time complexity is $O(n)$. The space complexity is $O(1)$.

Variant: Let m be a positive integer and L a sorted singly linked list of integers. For each integer k, if k appears more than m times in L, remove all nodes from L containing k.

7.9 Implement cyclic right shift for singly linked lists

This problem is concerned with performing a cyclic right shift on a list.

Write a program that takes as input a singly linked list and a nonnegative integer k, and returns the list cyclically shifted to the right by k. See Figure 7.9 for an example of a cyclic right shift.

Hint: How does this problem differ from rotating an array?

Solution: A brute-force strategy is to right shift the list by one node k times. Each right shift by a single node entails finding the tail, and its predecessor. The tail is prepended to the current head, and its original predecessor's successor is set to null to make it the new tail. The time complexity is $O(kn)$, and the space complexity is $O(1)$, where n is the number of nodes in the list.

Note that k may be larger than n. If so, it is equivalent to shift by k mod n, so we assume $k < n$. The key to improving upon the brute-force approach is to use the fact that linked lists can be cut and the sublists reassembled very efficiently. First we find the tail node t. Since the successor of the tail is the original head, we update t's successor. The original head is to become the kth node from the start of the new list. Therefore, the new head is the $(n - k)$th node in the initial list.

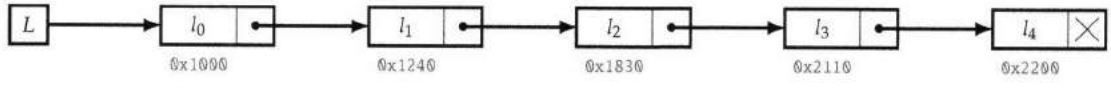
```
def cyclically_right_shift_list(L, k):
    if not L:
```

```
    return L

# Computes the length of L and the tail.
tail, n = L, 1
while tail.next:
    n += 1
    tail = tail.next

k %= n
if k == 0:
    return L

tail.next = L  # Makes a cycle by connecting the tail to the head.
steps_to_new_head, new_tail = n - k, tail
while steps_to_new_head:
    steps_to_new_head -= 1
    new_tail = new_tail.next

new_head = new_tail.next
new_tail.next = None
return new_head
```

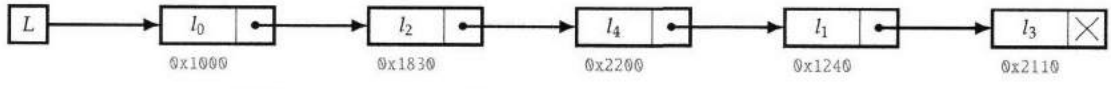
The time complexity is $O(n)$, and the space complexity is $O(1)$.

7.10 Implement even-odd merge

Consider a singly linked list whose nodes are numbered starting at 0. Define the even-odd merge of the list to be the list consisting of the even-numbered nodes followed by the odd-numbered nodes. The even-odd merge is illustrated in Figure 7.10.



(a) The initial list is L. The number in hex below a node indicates the memory address of that node.



(b) The even-odd merge of L—note that no new nodes have been allocated.

Figure 7.10: Even-odd merge example.

Write a program that computes the even-odd merge.

Hint: Use temporary additional storage.

Solution: The brute-force algorithm is to allocate new nodes and compute two new lists, one for the even and one for the odd nodes. The result is the first list concatenated with the second list. The time and space complexity are both $O(n)$.

However, we can avoid the extra space by reusing the existing list nodes. We do this by iterating through the list, and appending even elements to one list and odd elements to another list. We use an indicator variable to tell us which list to append to. Finally we append the odd list to the even list.

```
def even_odd_merge(L):
    if not L:
        return L

    even_dummy_head, odd_dummy_head = ListNode(0), ListNode(0)
    tails, turn = [even_dummy_head, odd_dummy_head], 0
    while L:
        tails[turn].next = L
        L = L.next
        tails[turn] = tails[turn].next
        turn ^= 1  # Alternate between even and odd.
    tails[1].next = None
    tails[0].next = odd_dummy_head.next
    return even_dummy_head.next
```

The time complexity is $O(n)$ and the space complexity is $O(1)$.

7.11 Test whether a singly linked list is palindromic

It is straightforward to check whether the sequence stored in an array is a palindrome. However, if this sequence is stored as a singly linked list, the problem of detecting palindromicity becomes more challenging. See Figure 7.1 on Page 82 for an example of a palindromic singly linked list.

Write a program that tests whether a singly linked list is palindromic.

Hint: It's easy if you can traverse the list forwards and backwards simultaneously.

Solution: A brute-force algorithm is to compare the first and last nodes, then the second and second-to-last nodes, etc. The time complexity is $O(n^2)$, where n is the number of nodes in the list. The space complexity is $O(1)$.

The $O(n^2)$ complexity comes from having to repeatedly traverse the list to identify the last, second-to-last, etc. Getting the first node in a singly linked list is an $O(1)$ time operation. This suggests paying a one-time cost of $O(n)$ time complexity to get the reverse of the second half of the original list, after which testing palindromicity of the original list reduces to testing if the first half and the reversed second half are equal. This approach changes the list passed in, but the reversed sublist can be reversed again to restore the original list.

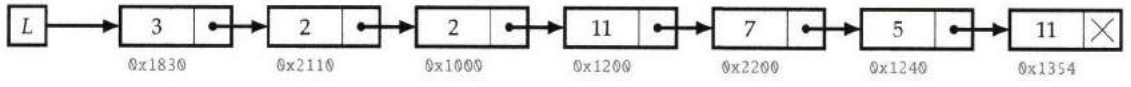
```
def is_linked_list_a_palindrome(L):
    # Finds the second half of L.
    slow = fast = L
    while fast and fast.next:
        fast, slow = fast.next.next, slow.next

    # Compares the first half and the reversed second half lists.
    first_half_iter, second_half_iter = L, reverse_linked_list(slow)
    while second_half_iter and first_half_iter:
        if second_half_iter.data != first_half_iter.data:
            return False
        second_half_iter, first_half_iter = (second_half_iter.next,
                                             first_half_iter.next)
    return True
```

The time complexity is $O(n)$. The space complexity is $O(1)$.

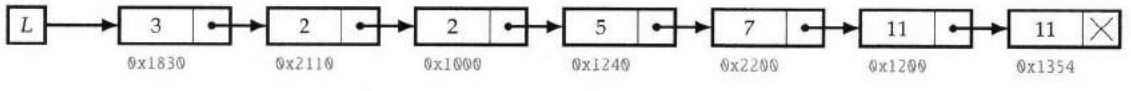
Variant: Solve the same problem when the list is doubly linked and you have pointers to the head and the tail.

7.12 Implement list pivoting

For any integer k, the pivot of a list of integers with respect to k is that list with its nodes reordered so that all nodes containing keys less than k appear before nodes containing k, and all nodes containing keys greater than k appear after the nodes containing k. See Figure 7.11 for an example of pivoting.



(a) A list to be pivoted.



(b) The result of pivoting the list in (a) with $k = 7$.

Figure 7.11: List pivoting.

Implement a function which takes as input a singly linked list and an integer k and performs a pivot of the list with respect to k. The relative ordering of nodes that appear before k, and after k, must remain unchanged; the same must hold for nodes holding keys equal to k.

Hint: Form the three regions independently.

Solution: A brute-force approach is to form three lists by iterating through the list and writing values into one of the three new lists based on whether the current value is less than, equal to, or greater than k. We then traverse the list from the head, and overwrite values in the original list from the less than, then equal to, and finally greater than lists. The time and space complexity are $O(n)$, where n is the number of nodes in the list.

A key observation is that we do not really need to create new nodes for the three lists. Instead we reorganize the original list nodes into these three lists in a single traversal of the original list. Since the traversal is in order, the individual lists preserve the ordering. We combine these three lists in the final step.

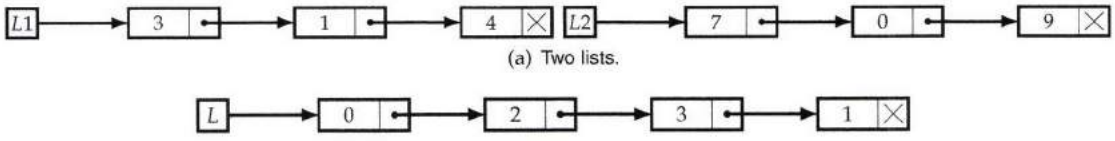
```
def list_pivoting(L, x):
    less_head = less_iter = ListNode()
    equal_head = equal_iter = ListNode()
    greater_head = greater_iter = ListNode()
    # Populates the three lists.
    while L:
        if L.data < x:
            less_iter.next = L
            less_iter = less_iter.next
        elif L.data == x:
            equal_iter.next = L
            equal_iter = equal_iter.next
        else:  # L.data > x.
            greater_iter.next = L
            greater_iter = greater_iter.next
        L = L.next
```

```
    # Combines the three lists.
    greater_iter.next = None
    equal_iter.next = greater_head.next
    less_iter.next = equal_head.next
    return less_head.next
```

The time to compute the three lists is $O(n)$. Combining the lists takes $O(1)$ time, yielding an overall $O(n)$ time complexity. The space complexity is $O(1)$.

7.13 Add list-based integers

A singly linked list whose nodes contain digits can be viewed as an integer, with the least significant digit coming first. Such a representation can be used to represent unbounded integers. This problem is concerned with adding integers represented in this fashion. See Figure 7.12 for an example.



(a) Two lists.

(b) The sum of the two lists in (a).

Figure 7.12: List-based interpretation of 413 + 907 = 1320.

Write a program which takes two singly linked lists of digits, and returns the list corresponding to the sum of the integers they represent. The least significant digit comes first.

Hint: First, solve the problem assuming no pair of corresponding digits sum to more than 9.

Solution: Note that we cannot simply convert the lists to integers, since the integer word length is fixed by the machine architecture, and the lists can be arbitrarily long.

Instead we mimic the grade-school algorithm, i.e., we compute the sum of the digits in corresponding nodes in the two lists. A key nuance of the computation is handling the carry-out. The algorithm continues processing input until both lists are exhausted and there is no remaining carry.

```
def add_two_numbers(L1, L2):
    place_iter = dummy_head = ListNode()
    carry = 0
    while L1 or L2 or carry:
        val = carry + (L1.data if L1 else 0) + (L2.data if L2 else 0)
        L1 = L1.next if L1 else None
        L2 = L2.next if L2 else None
        place_iter.next = ListNode(val % 10)
        carry, place_iter = val // 10, place_iter.next
    return dummy_head.next
```

The time complexity is $O(n + m)$ and the space complexity is $O(\max(n, m))$, where n and m are the lengths of the two lists.

Variant: Solve the same problem when integers are represented as lists of digits with the most significant digit comes first.