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

II Data Structures and Algorithms 13

2 Arrays 14
- 2.1 The Dutch national flag problem . . . 16
- 2.2 The Sudoku checker problem . . . 20
- 2.3 Compute the spiral ordering of a 2D array . . . 21
- 2.4 Rotate a 2D array . . . 24
- 2.5 Compute rows in Pascal's Triangle . . . 26

III Domain Specific Problems 27

3 Common Tools 28
- 3.1 Merging in a version control system . . . 28
- 3.2 Hooks . . . 30
- 3.3 SQL vs. NoSQL . . . 31
- 3.4 Normalization . . . 32
- 3.5 SQL design . . . 32

IV The Honors Class 34

4 Honors Class 35
- 4.1 Compute the greatest common divisor . . . 36
- 4.2 Find the first missing positive entry . . . 37
- 4.3 Buy and sell a stock k times . . . 38

V Notation and Index **40**

Notation **41**

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

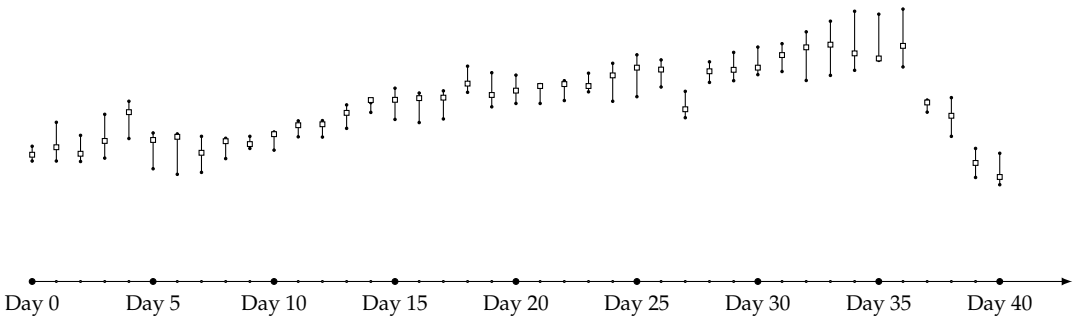


Figure 1: Share price as a function of time.

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 **??** on Page ??.

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–**??**, and are summarized in Table 1.1 on Page 7.

Chapter 1 is specifically concerned with preparation. Chapter **??** discusses how you should conduct yourself at the interview itself. Chapter **??** 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 **??**–**??** 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 **??**–**??** cover advanced algorithm design principles, such as dynamic programming and heuristics, as well as graphs. Chapter **??** 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 4 consists of a collection of more challenging problems.

Domain-specific knowledge is covered in Chapters **??**,**??**,**??**, and 3, 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 41. 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 43.

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 **??** 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–?? 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 ??–??. 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 4 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 4.

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 ??

- 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 ??

- 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 ??

- 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**
??	**??**	**??**	**??, ??**	**??**
2.1, **??**	**??**, 2.3	**??**, 2.2	**??, ??**	**??, ??, ??**
??	**??, ??**	**??, ??**	**??, ??**	**??, ??**
??	**??, ??**	**??, ??**	**??**	**??**
??	**??**	**??, ??**	**??, ??**	**??**
??	**??**	**??, ??**	**??**	**??, ??**
??	**??**	**??**	**??**	**??**
??	**??, ??**	**??, ??**	**??, ??**	**??, ??**
??	**??, ??**	**??, ??**	**??, ??**	**??**
??	**??**	**??**	**??, ??**	**??**
??	**??, ??**	**??, ??**	**??, ??**	**??, ??**
??	**??**	**??**	**??, ??**	**??, ??**
??	**??**	**??, ??**	**??, ??**	**??**
??	**??**	**??**	**??**	**??**
??	**??**	**??**	**??**	**??**
??	**??**	**??**	**??**	**??**
??	**??**	**??**	**??**	**??**

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 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 **??**).

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 ??. You should be especially be comfortable with lambda syntax, as well as the variable scoping rules for inner functions and lambdas.
- We use `collections.namedtuple`s 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 **??** on Page ?? 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.

Part II

Data Structures and Algorithms

Chapter

2 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 right), 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 2.1: 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.

2.1 The Dutch national flag problem

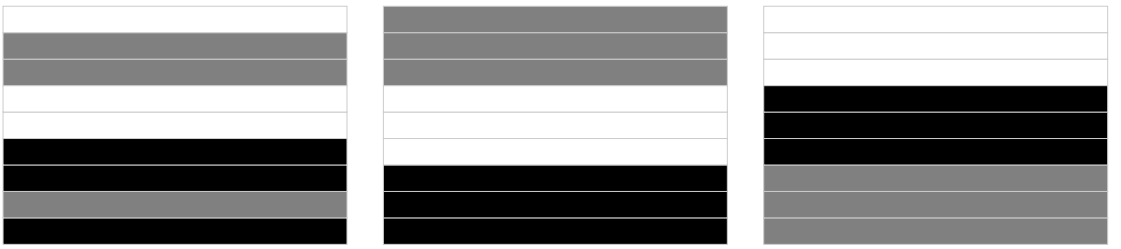
The quicksort algorithm for sorting arrays proceeds recursively—it selects an element (the "pivot"), reorders the array to make all the elements less than or equal to the pivot appear first, followed by all the elements greater than the pivot. The two subarrays are then sorted recursively.

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 2.1(b) is a valid partitioning for Figure 2.1(a). If gray precedes black and black precedes white, Figure 2.1(c) is a valid partitioning for Figure 2.1(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 2.1: 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 2.1(b) and 2.1(c) on Page 17 are valid answers for Figure 2.1(a) on Page 17. 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.

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 $A[i][j]$.

2.2 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 2.2(a); a complete solution is shown in Figure 2.2(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 2.2: 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
```

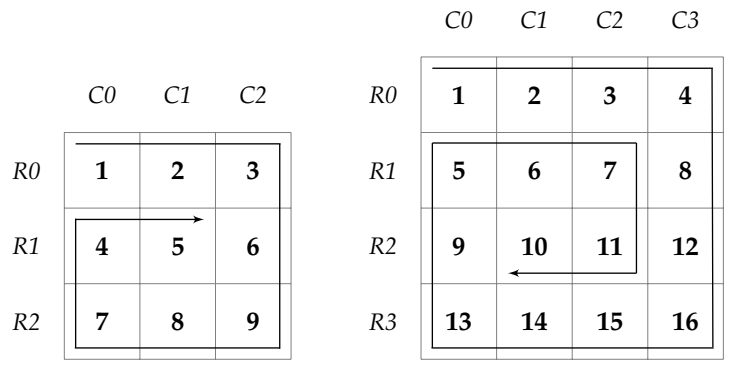
The time complexity of this algorithm for an $n \times n$ Sudoku grid with $\sqrt{n} \times \sqrt{n}$ subgrids is $O(n^2) + O(n^2) + O(n^2/(\sqrt{n})^2 \times (\sqrt{n})^2) = O(n^2)$; the terms correspond to the complexity to check n row constraints, the n column constraints, and the n subgrid constraints,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

Solution **??** on Page ?? describes how to solve Sudoku instances.

2.3 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 2.3(a) on the next page is $\langle 1, 2, 3, 6, 9, 8, 7, 4, 5 \rangle$. For Figure 2.3(b) on the following page, the spiral ordering is $\langle 1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7, 11, 10 \rangle$.

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



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

Figure 2.3: Spiral orderings. Column and row ids are specified above and to the left of the matrix, i.e., 1 is at entry $(0, 0)$.

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 2.3(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 2.3(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 that tracks the next element to process and the direction—left,right,up,down—to advance in. Think of the matrix as laid out on a $2D$ grid with X- and Y-axes. The pair (i, j) denotes the entry in Column i and Row j. Let (x, y) be the next element to process. Initially we move to the right (incrementing x until $(n-1, 0)$ is processed). Then we move down (incrementing y until $(n-1, n-1)$ is processed). Then we move left (decrementing x until $(0, n-1)$ is processed). Then we move up (decrementing y until $(0, 1)$ is processed). Note that we stop at 1, not 0, since $(0, 0)$ was already processed. 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 $(0, 1)$ we move to the right till we get to $(n-2, 1)$ (since $(n-2, 0)$ was already processed). This method is applied until all elements are processed.

```
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.

2.4 Rotate a 2D array

Image rotation is a fundamental operation in computer graphics. Figure 2.4 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 2.4: 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 2.4 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.

2.5 Compute rows in Pascal's Triangle

Figure 2.5 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 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.

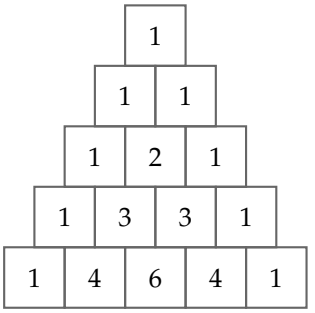


Figure 2.5: A Pascal triangle.

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.

Part III

Domain Specific Problems

Chapter

3 Common Tools

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages.

— "*The UNIX TimeSharing System*,"
D. Ritchie and K. Thompson, 1974

The problems in this chapter are concerned with tools: version control systems, scripting languages, build systems, databases, and the networking stack. Such problems are not commonly asked—expect them only if you are interviewing for a specialized role, or if you claim specialized knowledge, e.g., network security or databases. We emphasize these are vast subjects, e.g., networking is taught as a sequence of courses in a university curriculum. Our goal here is to give you a flavor of what you might encounter. Adnan's Advanced Programming Tools course material, available freely online, contains lecture notes, homeworks, and labs that may serve as a good resource on the material in this chapter.

Version control

Version control systems are a cornerstone of software development—every developer should understand how to use them in an optimal way.

3.1 Merging in a version control system

What is merging in a version control system? Specifically, describe the limitations of line-based merging and ways in which they can be overcome.

Solution:

Modern version control systems allow developers to work concurrently on a personal copy of the entire codebase. This allows software developers to work independently. The price for this merging: periodically, the personal copies need to be integrated to create a new shared version. There is the possibility that parallel changes are conflicting, and these must be resolved during the merge.

By far, the most common approach to merging is text-based. Text-based merge tools view software as text. Line-based merging is a kind of text-based merging, specifically one in which each line is treated as an indivisible unit. The merging is "three-way": the lowest common ancestor in the revision history is used in conjunction with the two versions. With line-based merging of text files, common text lines can be detected in parallel modifications, as well as text lines that have been

inserted, deleted, modified, or moved. Figure 3.1 on the following page shows an example of such a line-based merge.

One issue with line-based merging is that it cannot handle two parallel modifications to the same line: the two modifications cannot be combined, only one of the two modifications must be selected. A bigger issue is that a line-based merge may succeed, but the resulting program is broken because of syntactic or semantic conflicts. In the scenario in Figure 3.1 on the next page the changes made by the two developers are made to independent lines, so the line-based merge shows no conflicts. However, the program will not compile because a function is called incorrectly.

In spite of this, line-based merging is widely used because of its efficiency, scalability,and accuracy. A three-way, line-based merge tool in practice will merge 90% of the changed files without any issues. The challenge is to automate the remaining 10% of the situations that cannot be merged automatically.

Text-based merging fails because it does not consider any syntactic or semantic information.

Syntactic merging takes the syntax of the programming language into account. Text-based merge often yields unimportant conflicts such as a code comment that has been changed by different developers or conflicts caused by code reformatting. A syntactic merge can ignore all these: it displays a merge conflict when the merged result is not syntactically correct.

We illustrate syntactic merging with a small example:

```
if (n%2 == 0) then m = n/2;
```

Two developers change this code in a different ways, but with the same overall effect. The first updates it to

```
if (n%2 == 0) then m = n/2 else m = (n-1)/2;
```

and the second developer's update is

```
m = n/2;
```

Both changes to the same thing. However, a textual-merge will likely result in

```
m := n div 2 else m := (n-1)/2
```

which is syntactically incorrect. Syntactic merge identifies the conflict; it is the integrator's responsibility to manually resolve it, e.g., by removing the `else` portion.

Syntactic merge tools are unable to detect some frequently occurring conflicts. For example, in the merge of Versions *1a* and *1b* in Figure 3.1 on the following page will not compile, since the call to `sum(10)` has the wrong signature. Syntactic merge will not detect this conflict since the program is still syntactically correct. The conflict is a semantic conflict, specifically, a static semantic conflict, since it is detectable at compile-time. (The compile will return something like "function argument mismatch error".)

Technically, syntactic merge algorithms operate on the parse trees of the programs to be merged. More powerful static semantic merge algorithms have been proposed that operate on a graph representation of the programs, wherein definitions and usage are linked, making it easier to identify mismatched usage.

Static semantic merging also has shortcomings. For example, suppose `Point` is a class representing $2D$-points using Cartesian coordinates, and that this class has a distance function that returns $\sqrt{x^2 + y^2}$. Suppose Alice checks out the project, and subclasses `Point` to create a class that supports polar coordinates, and uses `Points`'s distance function to return the radius. Concurrently, Bob checks out the project and changes `Point`'s distance function to return $|x| + |y|$. Static semantic

merging reports no conflicts: the merged program compiles without any trouble. However the behavior is not what was expected.

Both syntactic merging and semantic merging greatly increase runtimes, and are very limiting since they are tightly coupled to specific programming languages. In practice, line-based merging is used in conjunction with a small set of unit tests (a "smoke suite") invoked with a pre-commit hook script. Compilation finds syntax and static semantics errors, and the tests (hopefully) identify deeper semantic issues.

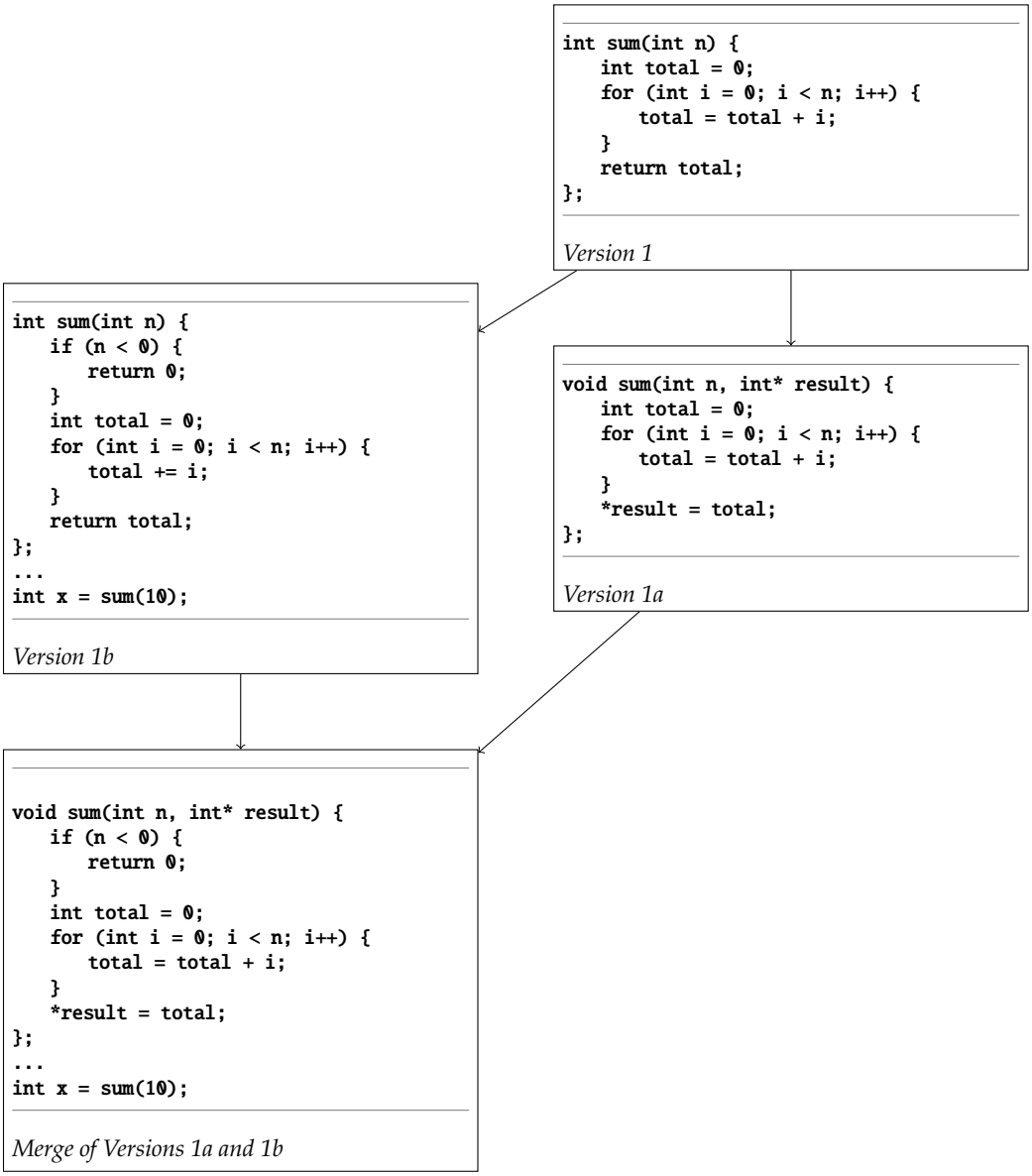


Figure 3.1: An example of a 3-way line based merge.

3.2 Hooks

What are hooks in a version control system? Describe their applications.

Solution: Version control systems such as git and SVN allow users to define actions to take around events such as commits, locking and unlocking files, and altering revision properties. These actions are specified as executable programs known as hook scripts. When the version control system gets to these events, it will check for the existence of a corresponding hook script, and execute it if it exists.

Hook scripts have access to the action as it is taking place, and are passed corresponding command-line arguments. For example, the pre-commit hook is passed the repository path and the transaction ID for the currently executing commit. If a hook script returns a nonzero exit code, the version control system aborts the action, returning the script's standard error output to the user.

The following hook scripts are used most often:

- *pre-commit:* Executed before a change is committed to the repository. Often used to check log messages, to format files, run a test suite, and to perform custom security or policy checking.
- *post-commit:* Executed once the commit has completed. Often used to inform users about a completed commit, for example by sending an email to the team, or update the bug tracking system.

As a rule, hooks should never alter the content of the transaction. At the very least, it can surprise a new developer; in the worst-case, the change can result in buggy code.

Database

Most software systems today interact with databases, and it's important for developers to have basic knowledge of databases.

3.3 SQL vs. NoSQL

Contrast SQL and NoSQL databases.

Solution: A relational database is a set of tables (also known as relations). Each table consists of rows. A row is a set of columns (also known as fields or attributes), which could be of various types (integer, float, date, fixed-size string, variable-size string, etc.). SQL is the language used to create and manipulate such databases. MySQL is a popular relational database.

A NoSQL database provides a mechanism for storage and retrieval of data which is modeled by means other than the tabular relations used in relational databases. MongoDB is a popular NoSQL database. The analog of a table in MongoDB is a collection, which consists of documents. Collections do not enforce a schema. Documents can be viewed as hash maps from fields to values. Documents within a collection can have different fields.

A key benefit of NoSQL databases is simplicity of design: a field can trivially be added to new documents in a collection without this affecting documents already in the database. This makes NoSQL a popular choice for startups which have an agile development environment.

The data structures used by NoSQL databases, e.g., key-value pairs in MongoDB, are different from those used by default in relational databases, making some operations faster in NoSQL. NoSQL databases are typically much simpler to scale horizontally, i.e., to spread documents from a collection across a cluster of machines.

A key benefit of relational databases include support for ACID transactions. ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. For example, there is no way in MongoDB to remove an item from inventory and add it to a customer's order as an atomic operation.

Another benefit of relational database is that their query language, SQL, is declarative, and relatively simple. In contrast, complex queries in a NoSQL database have to be implemented programmatically. (It's often the case that the NoSQL database will have some support for translating SQL into the equivalent NoSQL program.)

3.4 Normalization

What is database normalization? What are its advantages and disadvantages?
Solution: Database normalization is the process of organizing the columns and tables of a relational database to minimize data redundancy. Specifically, normalization involves decomposing a table into less redundant tables without losing information, thereby enforcing integrity and saving space.

The central idea is the use of "foreign keys". A foreign key is a field (or collection of fields) in one table that uniquely identifies a row of another table. In simpler words, the foreign key is defined in a second table, but it refers to the unique key in the first table. For example, a table called Employee may use a unique key called employee_id. Another table called Employee Details has a foreign key which references employee_id in order to uniquely identify the relationship between both the tables. The objective is to isolate data so that additions, deletions, and modifications of an attribute can be made in just one table and then propagated through the rest of the database using the defined foreign keys.

The primary drawback of normalization is performance—often, a large number of joins are required to recover the records the application needs to function.

3.5 SQL design

Write SQL commands to create a table appropriate for representing students as well as SQL commands that illustrate how to add, delete, and update students, as well as search for students by GPA with results sorted by GPA.

Then add a table of faculty advisors, and describe how to model adding an advisor to each student. Write a SQL query that returns the students who have an advisor with a specific name.
Solution: Natural fields for a student are name, birthday, GPA, and graduating year. In addition, we would like a unique key to serve as an identifier. A SQL statement to create a student table would look like the following: `CREATE TABLE students (id INT AUTO_INCREMENT, name VARCHAR(30), birthday DATE, gpa FLOAT, graduate_year INT, PRIMARY KEY(id));`. Here are examples of SQL statements for updating the students table.

- Add a student
  - `INSERT INTO students(name, birthday, gpa, graduate_year) VALUES ('Cantor', '1987-10-22', 3.9, 2009);`
- Delete students who graduated before 1985.
  - `DELETE FROM students WHERE graduate_year < 1985;`
- Update the GPA and graduating year of the student with id 28 to 3.14 and 2015, respectively.
  - `UPDATE students SET gpa = 3.14, graduate_year = 2015 WHERE id = 28;`

The following query returns students with a GPA greater than 3.5, sorted by GPA. It includes each student's name, GPA, and graduating year. `SELECT gpa, name, graduate_year FROM students WHERE gpa > 3.5 ORDER BY gpa DESC;`

Now we turn to the second part of the problem. Here is a sample table for faculty advisers.

id	*name*	*title*
1	Church	Dean
2	Tarski	Professor

We add a new column advisor_id to the students table. This is the foreign key on the preceding page. As an example, here's how to return students with GPA whose advisor is Church: `SELECT s.name, s.gpa FROM students s, advisors p WHERE s.advisor_id = p.id AND p.name = 'Church';`.

Variant: What is a SQL join? Suppose you have a table of courses and a table of students. How might a SQL join arise naturally in such a database?

Part IV

The Honors Class

Chapter

4 Honors Class

The supply of problems in mathematics is inexhaustible, and as soon as one problem is solved numerous others come forth in its place.

— "*Mathematical Problems*,"
D. Hilbert, 1900

This chapter contains problems that are more difficult to solve than the ones presented earlier. Many of them are commonly asked at interviews, albeit with the expectation that the candidate will not deliver the best solution.

There are several reasons why we included these problems:

- Although mastering these problems is not essential to your success, if you do solve them, you will have put yourself into a very strong position, similar to training for a race with ankle weights.
- If you are asked one of these questions or a variant thereof, and you successfully derive the best solution, you will have strongly impressed your interviewer. The implications of this go beyond being made an offer—your salary, position, and status will all go up.
- Some of the problems are appropriate for candidates interviewing for specialized positions, e.g., optimizing large scale distributed systems, machine learning for computational finance, etc. They are also commonly asked of candidates with advanced degrees.
- Finally, if you love a good challenge, these problems will provide it!

You should be happy if your interviewer asks you a hard question—it implies high expectations, and is an opportunity to shine, compared, for example, to being asked to write a program that tests if a string is palindromic.

Problems roughly follow the sequence of topics of the previous chapters, i.e., we begin with primitive types, and end with graphs. We recommend you solve these problems in a randomized order. The following problems are a good place to begin with: Problems 4.1, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**. White ninja () problems are though challenging, should be solvable by a good candidate given enough time. Black ninja () problems are exceptionally difficult, and are suitable for testing a candidate's response to stress, as described on Page**??**.

Intractability

Coding problems asked in interviews almost always have one (or more) solutions that are fast, e.g., $O(n)$ time complexity. Sometimes, you may be asked a problem that is computationally intractable—there may not exist an efficient algorithm for it. (Such problems are common in the real world.) There are a number of approaches to solving intractable problems:

- brute-force solutions, including dynamic programming, which have exponential time complexity, may be acceptable, if the instances encountered are small, or if the specific parameter that the complexity is exponential in is small;
- search algorithms, such as backtracking, branch-and-bound, and hill-climbing, which prune much of the complexity of a brute-force search;
- approximation algorithms which return a solution that is provably close to optimum;
- heuristics based on insight, common case analysis, and careful tuning that may solve the problem reasonably well;
- parallel algorithms, wherein a large number of computers can work on subparts simultaneously.

4.1 Compute the greatest common divisor 

The greatest common divisor (GCD) of nonnegative integers x and y is the largest integer d such that d divides x evenly, and d divides y evenly, i.e., $x \bmod d = 0$ and $y \bmod d = 0$. (When both x and y are 0, take their GCD to be 0.)

Design an efficient algorithm for computing the GCD of two nonnegative integers without using multiplication, division or the modulus operators.

Hint: Use case analysis: both even; both odd; one even and one odd.

Solution: The straightforward algorithm is based on recursion. If $x = y$, $\text{GCD}(x, y) = x$; otherwise, assume without loss of generality, that $x > y$. Then $\text{GCD}(x, y)$ is the $\text{GCD}(x - y, y)$.

The recursive algorithm based on the above does not use multiplication, division or modulus, but for some inputs it is very slow. As an example, if the input is $x = 2^n$, $y = 2$, the algorithm makes 2^{n-1} recursive calls. The time complexity can be improved by observing that the repeated subtraction amounts to division, i.e., when $x > y$, $\text{GCD}(x, y) = \text{GCD}(y, x \bmod y)$, but this approach uses integer division which was explicitly disallowed in the problem statement.

Here is a fast solution, which is also based on recursion, but does not use general multiplication or division. Instead it uses case-analysis, specifically, the cases of one, two, or none of the numbers being even.

An example is illustrative. Suppose we were to compute the GCD of 24 and 300. Instead of repeatedly subtracting 24 from 300, we can observe that since both are even, the result is $2 \times \text{GCD}(12, 150)$. Dividing by 2 is a right shift by 1, so we do not need a general division operation. Since 12 and 150 are both even, $\text{GCD}(12, 150) = 2 \times \text{GCD}(6, 75)$. Since 75 is odd, the GCD of 6 and 75 is the same as the GCD of 3 and 75, since 2 cannot divide 75. The GCD of 3 and 75 is the GCD of 3 and $75 - 3 = 72$. Repeatedly applying the same logic, $\text{GCD}(3, 72) = \text{GCD}(3, 36) = \text{GCD}(3, 18) = \text{GCD}(3, 9) = \text{GCD}(3, 6) = \text{GCD}(3, 3) = 3$. This implies $\text{GCD}(24, 300) = 2 \times 2 \times 3 = 12$.

More generally, the base case is when the two arguments are equal, in which case the GCD is that value, e.g., $\text{GCD}(12, 12) = 12$, or one of the arguments is zero, in which case the other is the GCD, e.g., $\text{GCD}(0, 6) = 6$.

Otherwise, we check if none, one, or both numbers are even. If both are even, we compute the GCD of the halves of the original numbers, and return that result times 2; if exactly one is even, we half it, and return the GCD of the resulting pair; if both are odd, we subtract the smaller from the

larger and return the GCD of the resulting pair. Multiplication by 2 is trivially implemented with a single left shift. Division by 2 is done with a single right shift.

```
def gcd(x, y):
    if x > y:
        return gcd(y, x)
    elif x == 0:
        return y
    elif not x & 1 and not y & 1:  # x and y are even.
        return gcd(x >> 1, y >> 1) << 1
    elif not x & 1 and y & 1:  # x is even, y is odd.
        return gcd(x >> 1, y)
    elif x & 1 and not y & 1:  # x is odd, y is even.
        return gcd(x, y >> 1)
    return gcd(x, y - x)  # Both x and y are odd.
```

Note that the last step leads to a recursive call with one even and one odd number. Consequently, in every two calls, we reduce the combined bit length of the two numbers by at least one, meaning that the time complexity is proportional to the sum of the number of bits in x and y, i.e., $O(\log x + \log y)$.

4.2 Find the first missing positive entry

Let A be an array of length n. Design an algorithm to find the smallest positive integer which is not present in A. You do not need to preserve the contents of A. For example, if $A = \langle 3, 5, 4, -1, 5, 1, -1 \rangle$, the smallest positive integer not present in A is 2.

Hint: First, find an upper bound for x.

Solution: A brute-force approach is to sort A and iterate through it looking for the first gap in the entries after we see an entry equal to 0. The time complexity is that of sorting, i.e., $O(n \log n)$.

Since all we want is the smallest positive number in A, we explore other algorithms that do not rely on sorting. We could store the entries in A in a hash table S (Chapter **??**), and then iterate through the positive integers $1, 2, 3, \ldots$ looking for the first one that is not in S. The time complexity is $O(n)$ to create S, and then $O(n)$ to perform the lookups, since we must find a missing entry by the time we get to $n + 1$ as there are only n entries. Therefore the time complexity is $O(n)$. The space complexity is $O(n)$, e.g., if the entries from A are all distinct positive integers.

The problem statement gives us a hint which we can use to reduce the space complexity. Instead of using an external hash table to store the set of positive integers, we can use A itself. Specifically, if A contains k between 1 and n, we set $A[k-1]$ to k. (We use $k - 1$ because we need to use all n entries, including the entry at index 0, which will be used to record the presence of 1.) Note that we need to save the presence of the existing entry in $A[k-1]$ if it is between 1 and n. Because A contains n entries, the smallest positive number that is missing in A cannot be greater than $n + 1$.

For example, let $A = \langle 3, 4, 0, 2 \rangle$, $n = 4$. we begin by recording the presence of 3 by writing it in $A[3-1]$; we save the current entry at index 2 by writing it to $A[0]$. Now $A = \langle 0, 4, 3, 2 \rangle$. Since 0 is outside the range of interest, we advance to $A[1]$, i.e., 4, which is within the range of interest. We write 4 in $A[4-1]$, and save the value at that location to index 1, and A becomes $\langle 0, 2, 3, 4 \rangle$. The value at $A[1]$ already indicates that a 2 is present, so we advance. The same holds for $A[2]$ and $A[3]$.

Now we make a pass through A looking for the first index i such that $A[i] \neq i + 1$; this is the smallest missing positive entry, which is 1 for our example.

```
def find_first_missing_positive(A):
    # Record which values are present by writing A[i] to index A[i] - 1 if
    # A[i] is between 1 and len(A), inclusive. We save the value at index A[i]
    # - 1 by swapping it with the entry at i. If A[i] is negative or greater
    # than n, we just advance i.
    for i in range(len(A)):
        while 1 <= A[i] <= len(A) and A[i] != A[A[i] - 1]:
            A[A[i] - 1], A[i] = A[i], A[A[i] - 1]

    # Second pass through A to search for the first index i such that A[i] !=
    # i+1, indicating that i + 1 is absent. If all numbers between 1 and
    # len(A) are present, the smallest missing positive is len(A) + 1.
    return next((i + 1 for i, a in enumerate(A) if a != i + 1), len(A) + 1)
```

The time complexity is $O(n)$, since we perform a constant amount of work per entry. Because we reuse A, the space complexity is $O(1)$.

4.3 Buy and sell a stock k times

This problem generalizes the buy and sell problem introduced on Page 1.

Write a program to compute the maximum profit that can be made by buying and selling a share k times over a given day range. Your program takes k and an array of daily stock prices as input.

Solution: Here is a straightforward algorithm. Iterate over j from 1 to k and iterate through A, recording for each index i the best solution for $A[0, i]$ with j pairs. We store these solutions in an auxiliary array of length n. The overall time complexity will be $O(kn^2)$; by reusing the arrays, we can reduce the additional space complexity to $O(n)$.

We can improve the time complexity to $O(kn)$, and the additional space complexity to $O(k)$ as follows. Define B_i^j to be the most money you can have if you must make $j - 1$ buy-sell transactions prior to i and buy at i. Define S_i^j to be the maximum profit achievable with j buys and sells with the jth sell taking place at i. Then the following mutual recurrence holds:

$$\begin{aligned} S_i^j &= A[i] + \max_{i' < i} B_{i'}^j \\ B_i^j &= \max_{i' < i} S_{i'}^{j-1} - A[i] \end{aligned}$$

The key to achieving an $O(kn)$ time bound is the observation that computing B and S requires computing $\max_{i' < i} B_{i'}^{j-1}$ and $\max_{i' < i} S_{i'}^{j-1}$. These two quantities can be computed in constant time for each i and j with a conditional update. In code:

```
def max_k_pairs_profits(prices, k):
    if not k:
        return 0.0
    elif 2 * k >= len(prices):
        return sum(max(0, b - a) for a, b in zip(prices[:-1], prices[1:]))
    min_prices, max_profits = [float('inf')] * k, [0] * k
    for price in prices:
        for i in reversed(list(range(k))):
```

```
        max_profits[i] = max(max_profits[i], price - min_prices[i])
        min_prices[i] = min(min_prices[i], price -
                            (0 if i == 0 else max_profits[i - 1]))
    return max_profits[-1]
```

Variant: Write a program that determines the maximum profit that can be obtained when you can buy and sell a single share an unlimited number of times, subject to the constraint that a buy must occur more than one day after the previous sell.

Part V

Notation and Index

Notation

To speak about notation as the only way that you can guarantee structure of course is already very suspect.

— E. S. Parker

We use the following convention for symbols, unless the surrounding text specifies otherwise:

A	k-dimensional array
L	linked list or doubly linked list
S	set
T	tree
G	graph
V	set of vertices of a graph
E	set of edges of a graph

Symbolism	Meaning
$(d_{k-1} \dots d_0)_r$	radix-r representation of a number, e.g., $(1011)_2$
$\log_b x$	logarithm of x to the base b; if b is not specified, $b = 2$
$\lvert S \rvert$	cardinality of set S
$S \setminus T$	set difference, i.e., $S \cap T'$, sometimes written as $S - T$
$\lvert x \rvert$	absolute value of x
$\lfloor x \rfloor$	greatest integer less than or equal to x
$\lceil x \rceil$	smallest integer greater than or equal to x
$\langle a_0, a_1, \dots, a_{n-1} \rangle$	sequence of n elements
$\sum_{R(k)} f(k)$	sum of all $f(k)$ such that relation $R(k)$ is true
$\min_{R(k)} f(k)$	minimum of all $f(k)$ such that relation $R(k)$ is true
$\max_{R(k)} f(k)$	maximum of all $f(k)$ such that relation $R(k)$ is true
$\sum_{k=a}^{b} f(k)$	shorthand for $\sum_{a \le k \le b} f(k)$
$\{a \mid R(a)\}$	set of all a such that the relation $R(a) =$ true
$[l, r]$	closed interval: $\{x \mid l \le x \le r\}$
$[l, r)$	left-closed, right-open interval: $\{x \mid l \le x < r\}$
$\{a, b, \dots\}$	well-defined collection of elements, i.e., a set
A_i or $A[i]$	the ith element of one-dimensional array A
$A[i, j]$	subarray of one-dimensional array A consisting of elements at indices i to j inclusive
$A[i][j]$	the element in ith row and jth column of 2D array A

$A[i_1, i_2][j_1, j_2]$	2D subarray of 2D array A consisting of elements from i_1th to i_2th rows and from j_1th to j_2th column, inclusive
$\binom{n}{k}$	binomial coefficient: number of ways of choosing k elements from a set of n items
$n!$	n-factorial, the product of the integers from 1 to n, inclusive
$O\big(f(n)\big)$	big-oh complexity of $f(n)$, asymptotic upper bound
$x \bmod y$	mod function
$x \oplus y$	bitwise-XOR function
$x \approx y$	x is approximately equal to y
`null`	pointer value reserved for indicating that the pointer does not refer to a valid address
$\emptyset$	empty set
∞	infinity: Informally, a number larger than any number.
$x \ll y$	much less than
$x \gg y$	much greater than
$\Rightarrow$	logical implication

Acknowledgments

Several of our friends, colleagues, and readers gave feedback. We would like to thank Taras Bobrovytsky, Senthil Chellappan, Yi-Ting Chen, Monica Farkash, Dongbo Hu, Jing-Tang Keith Jang, Matthieu Jeanson, Gerson Kurz, Danyu Liu, Hari Mony, Shaun Phillips, Gayatri Ramachandran, Ulises Reyes, Kumud Sanwal, Tom Shiple, Ian Varley, Shaohua Wan, Don Wong, and Xiang Wu for their input.

I, Adnan Aziz, thank my teachers, friends, and students from IIT Kanpur, UC Berkeley, and UT Austin for having nurtured my passion for programming. I especially thank my friends Vineet Gupta, Tom Shiple, and Vigyan Singhal, and my teachers Robert Solovay, Robert Brayton, Richard Karp, Raimund Seidel, and Somenath Biswas, for all that they taught me. My coauthor, Tsung-Hsien Lee, brought a passion that was infectious and inspirational. My coauthor, Amit Prakash, has been a wonderful collaborator for many years—this book is a testament to his intellect, creativity, and enthusiasm. I look forward to a lifelong collaboration with both of them.

I, Tsung-Hsien Lee, would like to thank my coauthors, Adnan Aziz and Amit Prakash, who give me this once-in-a-life-time opportunity. I also thank my teachers Wen-Lian Hsu, Ren-Song Tsay, Biing-Feng Wang, and Ting-Chi Wang for having initiated and nurtured my passion for computer science in general, and algorithms in particular. I would like to thank my friends Cheng-Yi He, Da-Cheng Juan, Chien-Hsin Lin, and Chih-Chiang Yu, who accompanied me on the road of savoring the joy of programming contests; and Kuan-Chieh Chen, Chun-Cheng Chou, Ray Chuang, Wilson Hong, Wei-Lun Hung, Nigel Liang, and Huan-Kai Peng, who give me valuable feedback on this book. Last, I would like to thank all my friends and colleagues at Google, Facebook, National Tsing Hua University, and UT Austin for the brain-storming on puzzles; it is indeed my greatest honor to have known all of you.

I, Amit Prakash, have my coauthor and mentor, Adnan Aziz, to thank the most for this book. To a great extent, my problem solving skills have been shaped by Adnan. There have been occasions in life when I would not have made it through without his help. He is also the best possible collaborator I can think of for any intellectual endeavor. I have come to know Tsung-Hsien through working on this book. He has been a great coauthor. His passion and commitment to excellence can be seen everywhere in this book. Over the years, I have been fortunate to have had great teachers at IIT Kanpur and UT Austin. I would especially like to thank my teachers Scott Nettles, Vijaya Ramachandran, and Gustavo de Veciana. I would also like to thank my friends and colleagues at Google, Microsoft, IIT Kanpur, and UT Austin for many stimulating conversations and problem solving sessions. Finally, and most importantly, I want to thank my family who have been a constant source of support, excitement, and joy all my life and especially during the process of writing this book.

Adnan Aziz — *Palo Alto, California*
Tsung-Hsien Lee — *Mountain View, California*
Amit Prakash — *Saratoga, California*
November 11, 2017