Welcome!

Thank you for joining us! This book powers our popular Data Structures and Algorithms online specialization on Coursera[1] and online MicroMasters program at edX[2]. We encourage you to sign up for a session and learn this material while interacting with thousands of other talented students from around the world. As you explore this book, you will find a number of active learning components that help you study the material at your own pace.

1. **PROGRAMMING CHALLENGES** ask you to implement the algorithms that you will encounter in one of programming languages that we support: `C`, `C++`, `Java`, `JavaScript`, `Python`, `Scala`, `C#`, `Haskell`, `Ruby`, and `Rust` (the last four programming languages are supported by Coursera only). These code challenges are embedded in our Coursera and edX online courses.

2. **ALGORITHMIC PUZZLES** provide you with a fun way to "invent" the key algorithmic ideas on your own! Even if you fail to solve some puzzles, the time will not be lost as you will better appreciate the beauty and power of algorithms. These puzzles are also embedded in our Coursera and edX online courses.

3. **EXERCISE BREAKS** offer "just in time" assessments testing your understanding of a topic before moving to the next one.

4. **STOP and THINK** questions invite you to slow down and contemplate the current material before continuing to the next topic.

[1] www.coursera.org/specializations/data-structures-algorithms
[2] www.edx.org/micromasters/ucsandiegox-algorithms-and-data-structures

Contents

About This Book

I find that I don't understand things unless I try to program them.
—Donald E. Knuth, *The Art of Computer Programming, Volume 4*

There are many excellent books on Algorithms — why in the world we would write another one???

Because we feel that while these books excel in introducing algorithmic ideas, they have not yet succeeded in teaching you how to implement algorithms, the crucial computer science skill.

Our goal is to develop an *Intelligent Tutoring System* for learning algorithms through programming that can compete with the best professors in a traditional classroom. This *MOOC book* is the first step towards this goal written specifically for our Massive Open Online Courses (MOOCs) forming a specialization *"Algorithms and Data Structures"* on Coursera platform[3] and a microMasters program on edX platform[4]. Since the launch of our MOOCs in 2016, hundreds of thousand students enrolled in this specialization and tried to solve more than hundred algorithmic programming challenges to pass it. And some of them even got offers from small companies like Google after completing our specialization!

In the last few years, some professors expressed concerns about the pedagogical quality of MOOCs and even called them the "junk food of education." In contrast, we are among the growing group of professors who believe that traditional classes, that pack hundreds of students in a single classroom, represent junk food of education. In a large classroom, once a student takes a wrong turn, there are limited opportunities to ask a question, resulting in a *learning breakdown*, or the inability to progress further without individual guidance. Furthermore, the majority of time a student invests in an Algorithms course is spent completing assignments outside the classroom. That is why we stopped giving lectures in our offline classes (and we haven't got fired yet :-). Instead, we give *flipped classes* where students watch our recorded lectures, solve algorithmic puzzles, complete programming challenges using our automated homework checking system before the class, and come to class prepared to discuss their learning

[3] www.coursera.org/specializations/data-structures-algorithms
[4] www.edx.org/micromasters/ucsandiegox-algorithms-and-data-structures

breakdowns with us.

When a student suffers a learning breakdown, that student needs immediate help in order to proceed. Traditional textbooks do not provide such help, but our automated grading system described in this MOOC book does! Algorithms is a unique discipline in that students' ability to program provides the opportunity to automatically check their knowledge through coding challenges. These coding challenges are far superior to traditional quizzes that barely check whether a student fell asleep. Indeed, to implement a complex algorithm, the student must possess a deep understanding of its underlying algorithmic ideas.

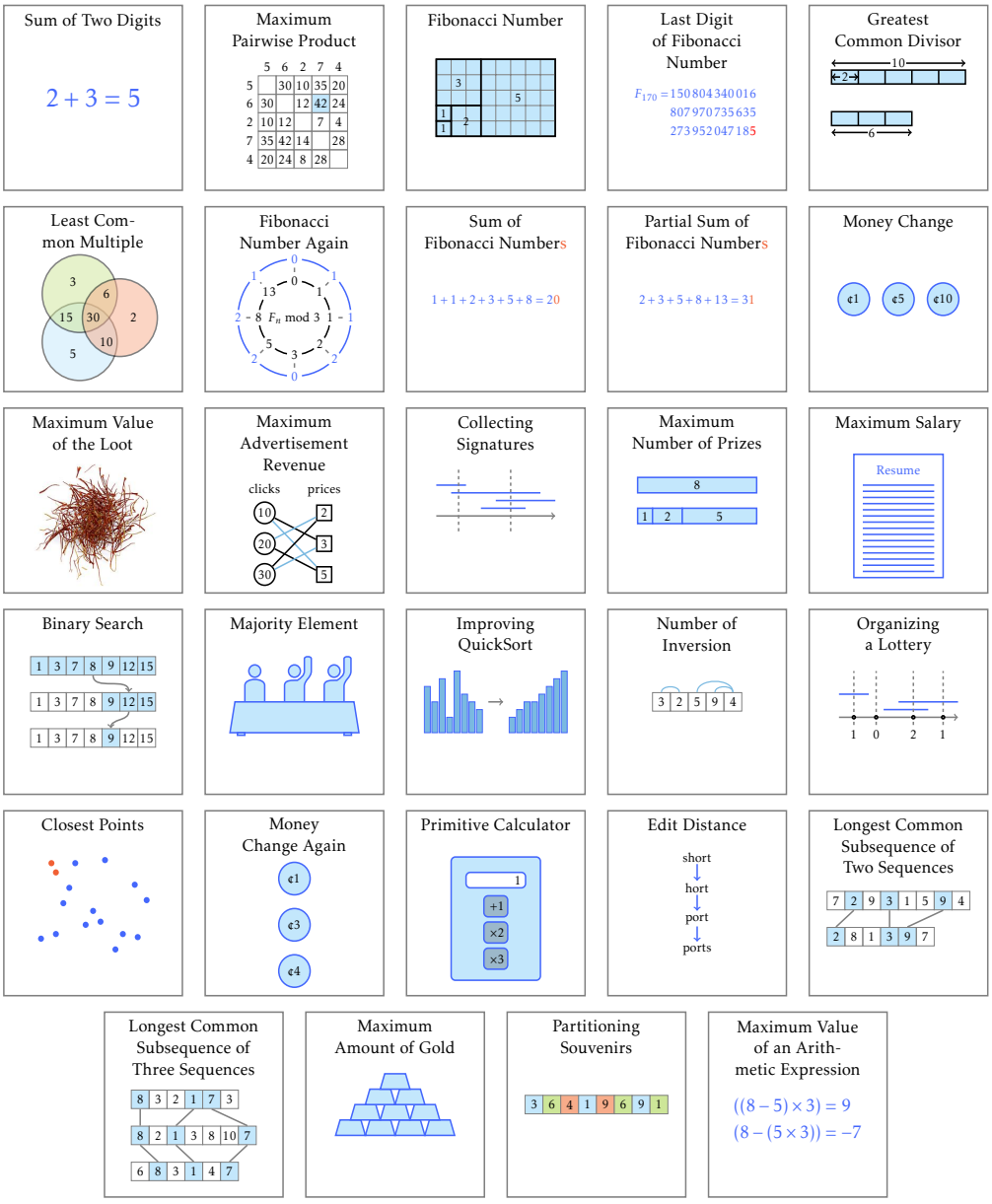
We believe that a large portion of grading in thousands of Algorithms courses taught at various universities each year can be consolidated into a single automated system available at all universities. It did not escape our attention that many professors teaching algorithms have implemented their own custom-made systems for grading student programs, an illustration of academic inefficiency and lack of cooperation between various instructors. Our goal is to build a repository of algorithmic programming challenges, thus allowing professors to focus on teaching. We have already invested thousands of hours into building such a system and thousands students in our MOOCs tested it. Below we briefly describe how it works.

When you face a programming challenge, your goal is to implement a fast and memory-efficient algorithm for its solution. Solving programming challenges will help you better understand various algorithms and may even land you a job since many high-tech companies ask applicants to solve programming challenges during the interviews. Your implementation will be checked automatically against many carefully selected tests to verify that it always produces a correct answer and fits into the time and memory constrains. Our system will teach you to write programs that work correctly on all of our test datasets rather than on some of them. This is an important skill since failing to thoroughly test your programs leads to undetected bugs that frustrate your boss, your colleagues, and, most importantly, users of your programs.

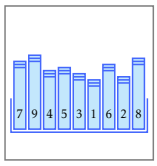
You maybe wondering why it took thousands of hours to develop such a system. First, we had to build a Compendium of Learning Breakdowns for each programming challenge, 10–15 most frequent errors that students make while solving it. Afterwards, we had to develop test cases for each learning breakdown in each programming challenge, over 20 000 test cases for just 100 programming challenges in our specialization.

Programming Challenges and Algorithmic Puzzles

This edition introduces basic algorithmic techniques using 29 programming challenges represented as icons below:

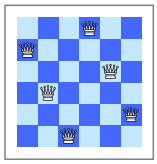


You are also welcome to solve the following algorithmic puzzles available at http://dm.compsciclub.ru/app/list:

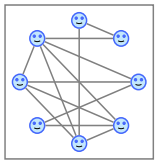


Book Sorting. Rearrange books on the shelf (in the increasing order of heights) using minimum number of swaps.

Map Coloring. Use minimum number of colors such that neighboring countries are assigned different colors and each country is assigned a single color.

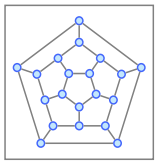


Eight Queens. Place eight queens on the chessboard such that no two queens attack each other (a queen can move horizontally, vertically, or diagonally).

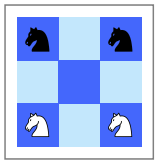


Clique Finding. Find the largest group of mutual friends (each pair of friends is represented by an edge).

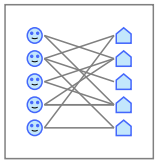
Hanoi Towers. Move all disks from one peg to another using a minimum number of moves. In a single move, you can move a top disk from one peg to any other peg provided that you don't place a larger disk on the top of a smaller disk.



Icosian Game. Find a cycle visiting each node exactly once.



Guarini Puzzle. Exchange the places of the white knights and the black knights. Two knights are not allowed to occupy the same cell of the chess board.



Room Assignment. Place each student in one of her/his preferable rooms in a dormitory so that each room is occupied by a single student (preferable rooms are shown by edges).

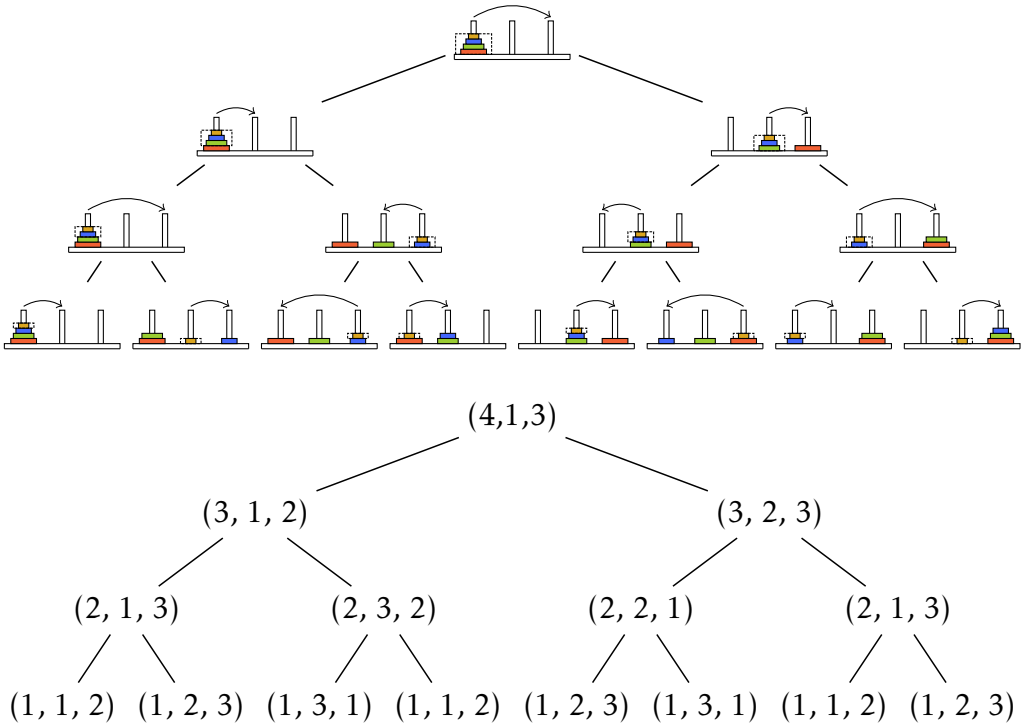


Figure 2.2: The recursion tree for a call to HanoiTowers$(4,1,3)$, which solves the Towers of Hanoi problem of size 4. At each point in the tree, (i,j,k) stands for HanoiTowers(i,j,k).

3.2.5 Stress Testing

We will now introduce *stress testing*—a technique for generating thousands of tests with the goal of finding a test case for which your solution fails.

A stress test consists of four parts:

1. Your implementation of an algorithm.
2. An alternative, trivial and slow, but correct implementation of an algorithm for the same problem.
3. A random test generator.
4. An infinite loop in which a new test is generated and fed into both implementations to compare the results. If their results differ, the test and both answers are output, and the program stops, otherwise the loop repeats.

The idea behind stress testing is that two correct implementations should give the same answer for each test (provided the answer to the problem is unique). If, however, one of the implementations is incorrect, then there exists a test on which their answers differ. The only case when it is not so is when there is the same mistake in both implementations, but that is unlikely (unless the mistake is somewhere in the input/output routines which are common to both solutions). Indeed, if one solution is correct and the other is wrong, then there exists a test case on which they differ. If both are wrong, but the bugs are different, then most likely there exists a test on which two solutions give different results.

Here is the the stress test for MaxPairwiseProductFast using MaxPairwiseProductNaive as a trivial implementation:

stars, etc. If you are generating integers, try generating both prime and composite numbers.

3.3.5 Submitting to the Grading System

When you are done with testing, submit your program to the grading system! Go to the submission page, create a new submission, and upload a file with your program (make sure to upload a source file rather than an executable). The grading system then compiles your program and runs it on a set of carefully constructed tests to check that it outputs a correct result for all tests and that it fits into the time and memory limits. The grading usually takes less than a minute, but in rare cases, when the servers are overloaded, it might take longer. Please be patient. You can safely leave the page when your solution is uploaded.

As a result, you get a feedback message from the grading system. You want to see the "`Good job!`" message indicating that your program passed all the tests. The messages "`Wrong answer`", "`Time limit exceeded`", "`Memory limit exceeded`" notify you that your program failed due to one of these reasons. If you program fails on one of the first two test cases, the grader will report this to you and will show you the test case and the output of your program. This is done to help you to get the input/output format right. In all other cases, the grader will *not* show you the test case where your program fails.

3.4 Good Programming Practices

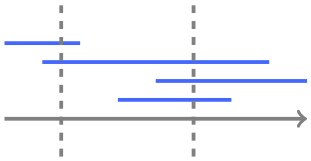
Programming is an art of not making off-by-one errors. In this section, we will describe some good practices for software implementation that will help you to avoid off-by-one bugs (OBOBs) and many other common programming pitfalls. Sticking to these good practices will help you to write a reliable, compact, readable, and debuggable code.

Stick to a specific code style.
Mixing various code styles in your programs make them less readable. See https://en.wikipedia.org/wiki/Programming_style to select your favorite code style.

5.4 Collecting Signatures

Covering Segments by Points Problem
Find the minimum number of points needed to cover all given segments on a line.

Input: A sequence of n segments $[a_1, b_1], \ldots, [a_n, b_n]$ on a line.
Output: A set of points of minimum size such that each segment $[a_i, b_i]$ contains a point, i.e., there exists a point x such that $a_i \le x \le b_i$.



You are responsible for collecting signatures from all tenants in a building. For each tenant, you know a period of time when he or she is at home. You would like to collect all signatures by visiting the building as few times as possible. For simplicity, we assume that when you enter the building, you instantly collect the signatures of all tenants that are in the building at that time.

Input format. The first line of the input contains the number n of segments. Each of the following n lines contains two integers a_i and b_i (separated by a space) defining the coordinates of endpoints of the i-th segment.

Output format. The minimum number m of points on the first line and the integer coordinates of m points (separated by spaces) on the second line. You can output the points in any order. If there are many such sets of points, you can output any set.

Constraints. $1 \le n \le 100$; $0 \le a_i \le b_i \le 10^9$ for all i.