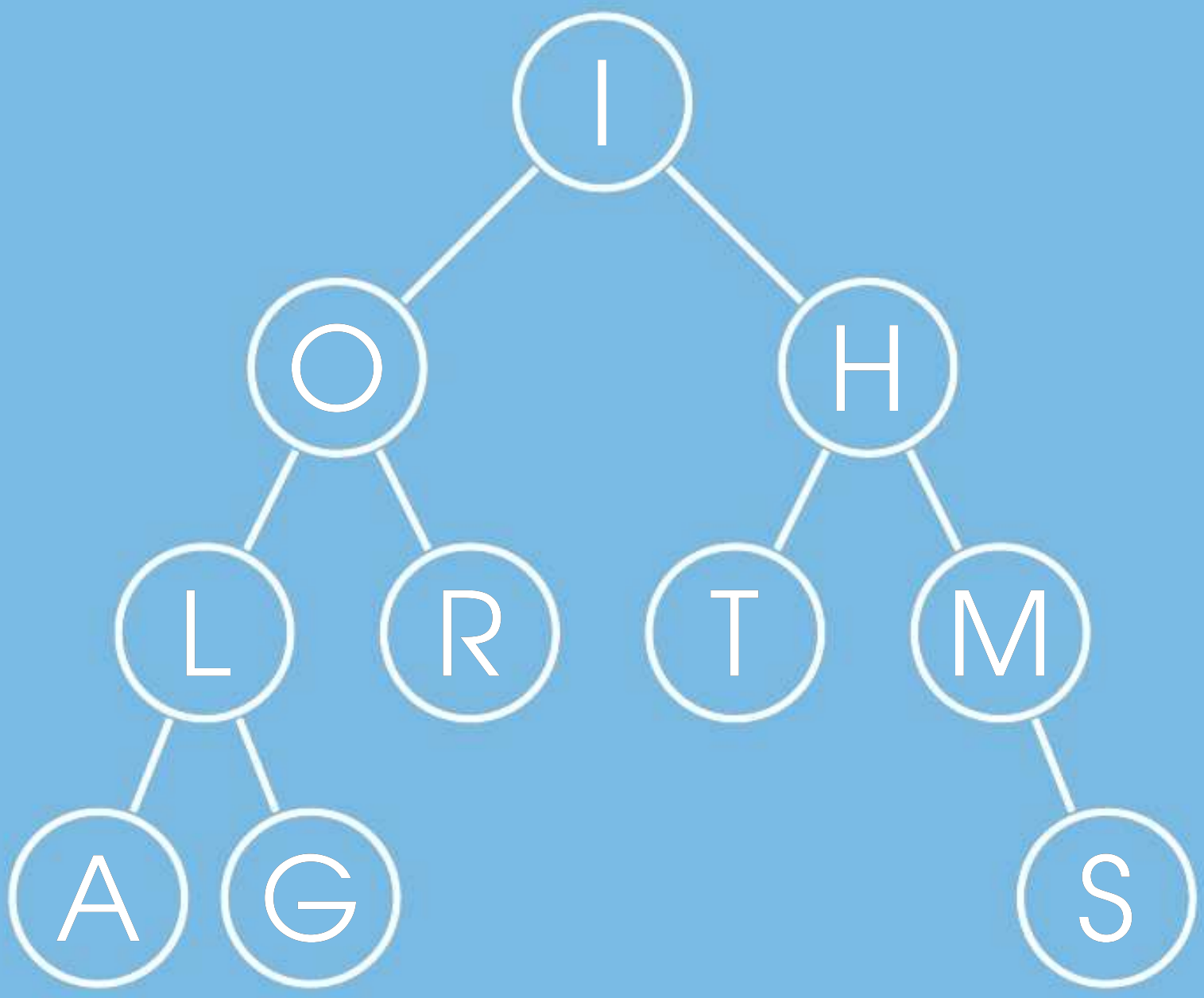
LEARNING ALGORITHMS THROUGH PROGRAMMING AND PUZZLE SOLVING



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

Learning Algorithms Through Programming and Puzzle Solving

Alexander S. Kulikov and Pavel Pevzner

Active Learning Technologies

ISBN: 978-0-9996762-0-2

Active Learning Technologies

Address:
3520 Lebon Drive
Suite 5208
San Diego, CA 92122, USA

To my parents. — A.K.

To my family. — P.P.

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.

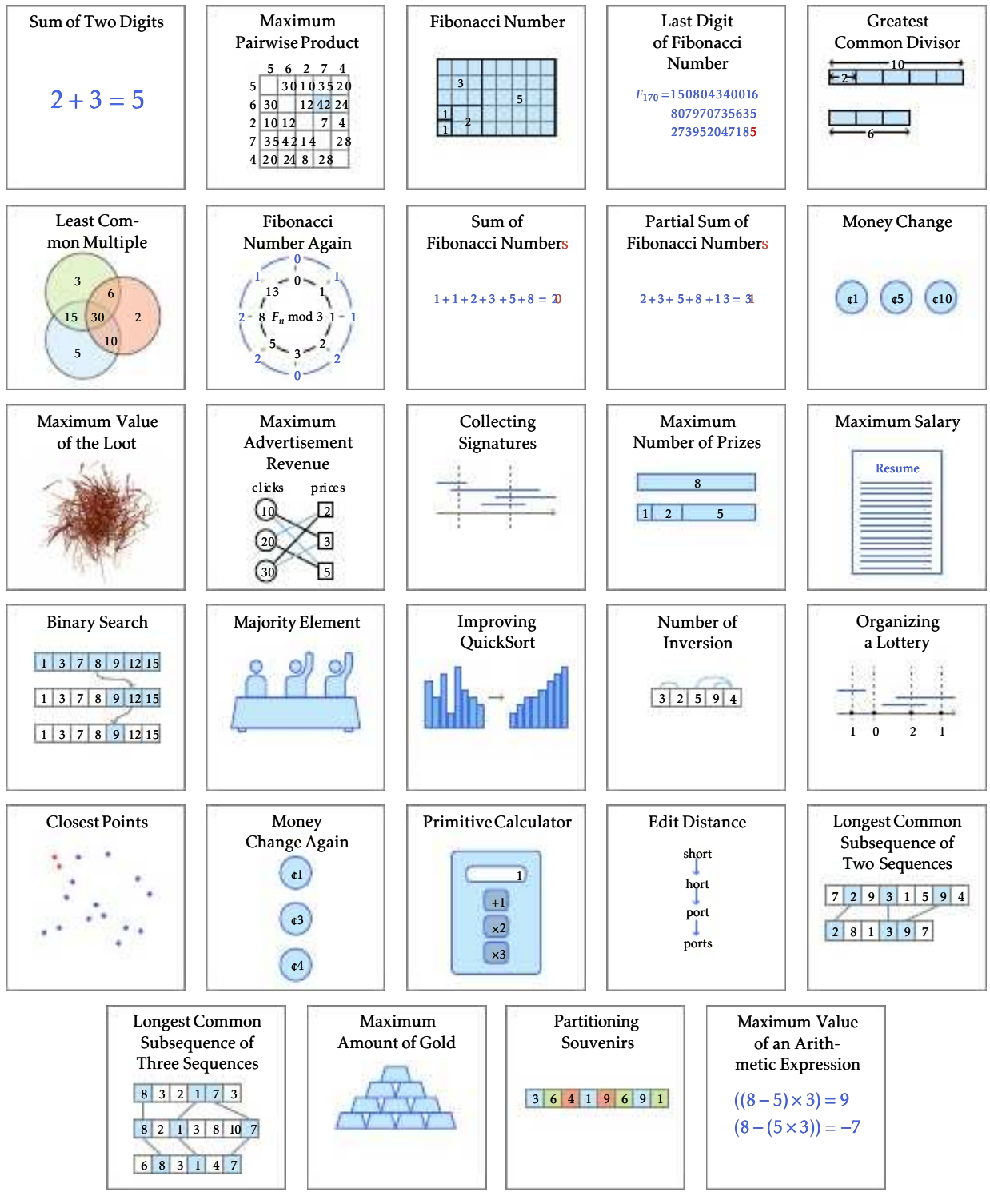
When you face a programming challenge, your goal is to implement a fast and memory-efficient algorithm for its solution. Solving programming challenges will help you better understand various algorithms and may even land you a job since many high-tech companies ask applicants to solve programming challenges 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.

We encourage you to sign up for our *Algorithms and Data Structures* specialization on Coursera or MicroMasters program on edX and start interacting with thousands of talented students from around the world who are learning algorithms. Thank you for joining us!

Programming Challenges and Algorithmic Puzzles

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

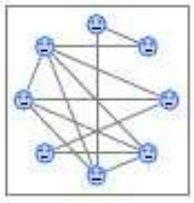


You are also welcome to solve the following algorithmic puzzles available at http://dm.compsciclub.ru/app/list:

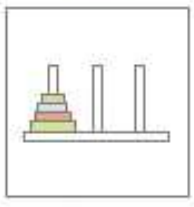
Book Sorting. Rearrange books on the shelf (in the increasing order of heights) using minimum number of swaps.

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

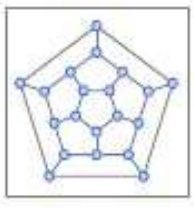
Eight Queens. Place eight queens on the chessboard such that no two queens attack each other (a queen can move horizontally, vertically, or diagonally).



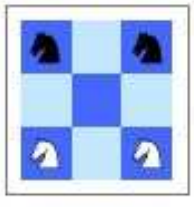
Clique Finding. Find the largest group of mutual friends (each pair of friends is represented by an edge).



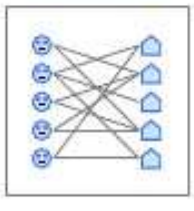
Hanoi Towers. Move all disks from one peg to another using a minimum number of moves. In a single move, you can move a top disk from one peg to any other peg provided that you don't place a larger disk on the top of a smaller disk.



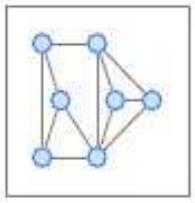
Icosian Game. Find a cycle visiting each node exactly once.



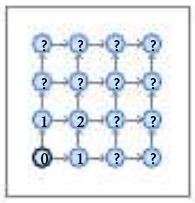
Guarini Puzzle. Exchange the places of the white knights and the black knights. Two knights are not allowed to occupy the same cell of the chess board.



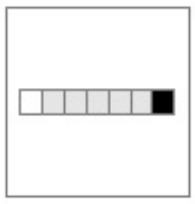
Room Assignment. Place each student in one of her/his preferable rooms in a dormitory so that each room is occupied by a single student (preferable rooms are shown by edges).



Tree Construction. Remove the minimum number of edges from the graph to make it acyclic.

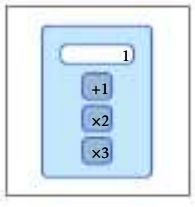


Number of Paths. Find out how many paths are there to get from the bottom left circle to any other circle and place this number inside the corresponding circle.

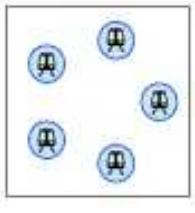


Black and White Squares. Use the minimum number of questions "What is the color of this square?" to find two neighboring squares of different colors. The leftmost square is white, the rightmost square is black, but the colors of all other squares are unknown.

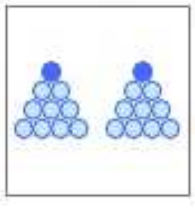
Twenty One Questions Game. Find an unknown integer $1 \le x \le N$ by asking the minimum number of questions "Is $x = y$?" (for any $1 \le y \le N$). Your opponent will reply either "Yes", or "$x < y$", or "$x > y$."



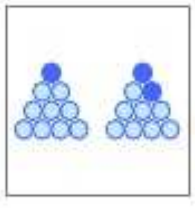
Antique Calculator. Find the minimum number of operations needed to get a positive integer n from the integer 1 using only three operations: add 1, multiply by 2, or multiply by 3.



Subway Lines. You are planning a subway system where the subway lines should not cross. Can you connect each pair of the five stations except for a single pair?



Two Rocks Game. There are two piles of ten rocks. In each turn, you and your opponent may either take one rock from a single pile, or one rock from both piles. Your opponent moves first and the player that takes the last rock wins the game. Design a winning strategy.



Three Rocks Game. There are two piles of ten rocks. In each turn, you and your opponent may take up to three rocks. Your opponent moves first and the player that takes the last rock wins the game. Design a winning strategy.

What Lies Ahead

Watch for our future editions that will cover the following topics.

- Data Structures
  - Arrays and Lists
  - Priority Queues
  - Disjoint Sets
  - Hash Tables
  - Binary Search Trees
- Algorithms on Graphs
  - Graphs Decomposition
  - Shortest Paths in Graphs
  - Minimum Spanning Trees
  - Shortest Paths in Real Life
- Algorithms on Strings
  - Pattern Matching
  - Suffix Trees
  - Suffix Arrays
  - Burrows–Wheeler Transform
- Advanced Algorithms and Complexity
  - Flows in Networks
  - Linear Programmings
  - NP-complete Problems
  - Coping with NP-completeness
  - Streaming Algorithms

Meet the Authors

Alexander S. Kulikov is a senior research fellow at Steklov Mathematical Institute of the Russian Academy of Sciences, Saint Petersburg, Russia and a lecturer at the Department of Computer Science and Engineering at University of California, San Diego, USA. He also directs the Computer Science Center in Saint Petersburg that provides free advanced computer science courses complementing the standard university curricula. Alexander holds a Ph. D. from Steklov Mathematical Institute. His research interests include algorithms and complexity theory. He co-authored online courses "Data Structures and Algorithms" and "Introduction to Discrete Mathematics for Computer Science" that are available at Coursera and edX.

Pavel Pevzner is Ronald R. Taylor Professor of Computer Science at the University of California, San Diego. He holds a Ph. D. from Moscow Institute of Physics and Technology, Russia and an Honorary Degree from Simon Fraser University. He is a Howard Hughes Medical Institute Professor (2006), an Association for Computing Machinery Fellow (2010), an International Society for Computational Biology Fellow (2012), and a Member of the the Academia Europaea (2016). He has authored the textbooks Computational Molecular Biology: An Algorithmic Approach (2000), An Introduction to Bioinformatics Algorithms (2004) (jointly with Neil Jones), and Bioinformatics Algorithms: An Active Learning Approach (2014) (jointly with Phillip Compeau). He co-authored online courses "Data Structures and Algorithms", "Bioinformatics", and "Analyze Your Genome!" that are available at Coursera and edX.

Meet Our Online Co-Instructors

Daniel Kane is an associate professor at the University of California, San Diego with a joint appointment between the Department of Computer Science and Engineering and the Department of Mathematics. He has diverse interests in mathematics and theoretical computer science, though most of his work fits into the broad categories of number theory, complexity theory, or combinatorics.

Michael Levin is an Associate Professor at the Computer Science Department of Higher School of Economics, Moscow, Russia and the Chief Data Scientist at the Yandex.Market, Moscow, Russia. He also teaches Algorithms and Data Structures at the Yandex School of Data Analysis.

Neil Rhodes is a lecturer in the Computer Science and Engineering department at the University of California, San Diego and formerly a staff software engineer at Google. Neil holds a B.A. and M.S. in Computer Science from UCSD. He left the Ph.D. program at UCSD to start a company, Palomar Software, and spent fifteen years writing software, books on software development, and designing and teaching programming courses for Apple and Palm. He's taught Algorithms, Machine Learning, Operating Systems, Discrete Mathematics, Automata and Computability Theory, and Software Engineering at UCSD and Harvey Mudd College in Claremont, California.

Acknowledgments

This book was greatly improved by the efforts of a large number of individuals, to whom we owe a debt of gratitude.

Our co-instructors and partners in crime Daniel Kane, Michael Levin, and Neil Rhodes invested countless hours in the development of our online courses at Coursera and edX platforms.

Hundreds of thousands of our online students provided valuable feedback that led to many improvements in our MOOCs and this MOOC book. In particular, we are grateful to the mentors of the Algorithmic Toolbox class at Coursera: Ayoub Falah, Denys Diachenko, Kishaan Jeeveswaran, Irina Pinjaeva, Fernando Gonzales Vigil Richter, and Gabrio Secco.

We thank our colleagues who helped us with preparing programming challenges: Maxim Akhmedov, Roman Andreev, Gleb Evstropov, Nikolai Karpov, Sergey Poromov, Sergey Kopeliovich, Ilya Kornakov, Gennady Korotkevich, Paul Melnichuk, and Alexander Tiunov.

We are grateful to Anton Konev for leading the development of interactive puzzles as well as Anton Belyaev and Kirill Banaru for help with some of the puzzles.

We thank Alexey Kladov and Alexander Smal for help with the "Good Programming Practices" section of the book.

Randall Christopher brought to life our idea for the textbook cover.

Finally, our families helped us preserve our sanity when we were working on this MOOC book.

A. K. and P. P.
Saint Petersburg and San Diego
December 2017

Chapter 1: Algorithms and Complexity

This book presents programming challenges that will teach you how to design and implement algorithms. Solving a programming challenge is one of the best ways to understand an algorithm's design as well as to identify its potential weaknesses and fix them.

1.1 What Is an Algorithm?

Roughly speaking, an algorithm is a sequence of instructions that one must perform in order to solve a well-formulated problem. We will specify problems in terms of their *inputs* and their *outputs*, and the algorithm will be the method of translating the inputs into the outputs. A well-formulated problem is unambiguous and precise, leaving no room for misinterpretation.

After you designed an algorithm, two important questions to ask are: "Does it work correctly?" and "How much time will it take?" Certainly you would not be satisfied with an algorithm that only returned correct results half the time, or took 1000 years to arrive at an answer.

1.2 Pseudocode

To understand how an algorithm works, we need some way of listing the steps that the algorithm takes, while being neither too vague nor too formal. We will use *pseudocode*, a language computer scientists often use to describe algorithms. Pseudocode ignores many of the details that are required in a programming language, yet it is more precise and less ambiguous than, say, a recipe in a cookbook.

1.3 Problem Versus Problem Instance

A problem describes a class of computational tasks. A problem instance is one particular input from that class. To illustrate the difference between a problem and an instance of a problem, consider the following example. You find yourself in a bookstore buying a book for $4.23 which you pay

for with a $5 bill. You would be due 77 cents in change, and the cashier now makes a decision as to exactly how you get it. You would be annoyed at a fistful of 77 pennies or 15 nickels and 2 pennies, which raises the question of how to make change in the least annoying way. Most cashiers try to minimize the number of coins returned for a particular quantity of change. The example of 77 cents represents an instance of the Change Problem, which we describe below.

The example of 77 cents represents an instance of the Change Problem that assumes that there are d denominations represented by an array $c = (c_1, c_2, \ldots, c_d)$. For simplicity, we assume that the denominations are given in decreasing order of value. For example, $c = (25, 10, 5, 1)$ for United States denominations.

Change Problem
Convert some amount of money into given denominations, using the smallest possible number of coins.

Input: An integer *money* and an array of d denominations $c = (c_1, c_2, \ldots, c_d)$, in decreasing order of value ($c_1 > c_2 > \cdots > c_d$).
Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 \cdot i_1 + c_2 \cdot i_2 + \cdots + c_d \cdot i_d = money$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

The algorithm that is used by cashiers all over the world to solve this problem is simple:

```
CHANGE(money, c, d):
while money > 0:
  coin ← coin with the largest denomination that does not exceed money
  give coin with denomination coin to customer
  money ← money − coin
```

Here is a faster version of CHANGE:

```
CHANGE(money, c, d):
r ← money
for k from 1 to d:
  i_k ← ⌊r / c_k⌋
  r ← r − c_k · i_k
return (i_1, i_2, ..., i_d)
```

1.4 Correct Versus Incorrect Algorithms

We say that an algorithm is correct when it translates every input instance into the correct output. An algorithm is incorrect when there is at least one input instance for which the algorithm gives an incorrect output.

CHANGE is an incorrect algorithm! Suppose you were changing 40 cents into coins with denominations of $c_1 = 25$, $c_2 = 20$, $c_3 = 10$, $c_4 = 5$, and $c_5 = 1$. CHANGE would incorrectly return 1 quarter, 1 dime, and 1 nickel, instead of 2 twenty-cent pieces. As contrived as this may seem, in 1875 a twenty-cent coin existed in the United States. How sure can we be that CHANGE returns the minimal number of coins for the modern US denominations or for denominations in any other country?

To correct the CHANGE algorithm, we could consider every possible combination of coins with denominations $c_1, c_2, \ldots, c_d$ that adds to *money*, and return the combination with the fewest. We only need to consider combinations with $i_1 \leq money/c_1$ and $i_2 \leq money/c_2$ (in general, i_k should not exceed $money/c_k$), because we would otherwise be returning an amount of money larger than *money*. The pseudocode below uses the symbol $\sum$ that stands for summation: $\sum_{i=1}^{m} a_i = a_1 + a_2 + \cdots + a_m$. The pseudocode also uses the notion of "infinity" (denoted as ∞) as an initial value for *smallestNumberOfCoins*; there are a number of ways to carry this out in a real computer, but the details are not important here.

```
BRUTEFORCECHANGE(money, c, d):
smallestNumberOfCoins ← ∞
for each (i_1,...,i_d) from (0,...,0) to (money/c_1,...,money/c_d)
    valueOfCoins ← Σ_{k=1}^{d} i_k · c_k
    if valueOfCoins = M:
        numberOfCoins = Σ_{k=1}^{d} i_k
        if numberOfCoins < smallestNumberOfCoins:
            smallestNumberOfCoins ← numberOfCoins
            change ← (i_1, i_2,..., i_d)
return change
```

The second line iterates over every feasible combination $(i_1, \ldots, i_d)$ of the d indices, and stops when it has reached $(money/c_1, \ldots, money/c_d)$.

How do we know that BRUTEFORCECHANGE does not suffer from the same problem as CHANGE did, namely that it generates incorrect result

for some input instance?? Since BruteForceChange explores all feasible combinations of denominations, it will eventually come across an optimal solution and record it as such in the *change* array. Any combination of coins that adds to M must have at least as many coins as the optimal combination, so BruteForceChange will never overwrite *change* with a suboptimal solution.

So far we have answered only one of the two important algorithmic questions ("Does it work?", but not "How much time will it take?").

Stop and Think. How fast is BruteForceChange?

1.5 Fast Versus Slow Algorithms

Real computers require a certain amount of time to perform an operation such as addition, subtraction, or testing the conditions in a while loop. A supercomputer might take 10^{-10} second to perform an addition, while a calculator might take 10^{-5} second. Suppose that you had a computer that took 10^{-10} second to perform an elementary operation such as addition, and that you knew how many operations a particular algorithm would perform. You could estimate the running time of the algorithm simply by taking the product of the number of operations and the time per operation. However, computers are constantly improving, leading to a decreasing time per operation, so your notion of the running time would soon be outdated. Rather than computing an algorithm's running time on every computer, we rely on the total number of operations that the algorithm performs to describe its running time, since this is an attribute of the algorithm, and not an attribute of the computer you happen to be using.

Unfortunately, determining how many operations an algorithm will perform is not always easy. If we know how to compute the number of basic operations that an algorithm performs, then we have a basis to compare it against a different algorithm that solves the same problem. Rather than tediously count every multiplication and addition, we can perform this comparison by gaining a high-level understanding of the growth of each algorithm's operation count as the size of the input increases.

Suppose an algorithm A performs $11n^3$ operations on an input of size n, and an algorithm B solves the same problem in $99n^2 + 7$ opera-

tions. Which algorithm, A or B, is faster? Although A may be faster than B for some small n (e.g., for n between 0 and 9), B will become faster for large n (e.g., for all $n > 10$). Since n^3 is, in some sense, a "faster-growing" function than n^2 with respect to n , the constants 11, 99, and 7 do not affect the competition between the two algorithms for large n. We refer to A as a *cubic* algorithm and to B as a *quadratic* algorithm, and say that A is less efficient than B because it performs more operations to solve the same problem when n is large. Thus, we will often be somewhat imprecise when we count operations of an algorithm—the behavior of algorithms on small inputs does not matter.

Let's estimate the number of operations BruteForceChange will take on an input instance of M cents, and denominations $(c_1, c_2, \ldots, c_d)$. To calculate the total number of operations in the for loop, we can take the approximate number of operations performed in each iteration and multiply this by the total number of iterations. Since there are roughly

$$\frac{money}{c_1} \times \frac{money}{c_2} \times \cdots \times \frac{money}{c_d}$$

iterations, the for loop performs on the order of $d \times \frac{money^d}{c_1 c_2 \cdots c_d}$ operations, which dwarfs the other operations of the algorithm.

This type of algorithm is often referred to as an *exponential* algorithm in contrast to quadratic, cubic, or other *polynomial* algorithms. The expression for the running time of exponential algorithms includes a term like n^d, where n and d are parameters of the problem (i.e., n and d may deliberately be made arbitrarily large by changing the input to the algorithm), while the running time of a polynomial algorithm is bounded by a term like n^k where k is a constant not related to the size of any parameters.

For example, an algorithm with running time n^1 (linear), n^2 (quadratic), n^3 (cubic), or even n^{2018} is polynomial. Of course, an algorithm with running time n^{2018} is not very practical, perhaps less so than some exponential algorithms, and much effort in computer science goes into designing faster and faster polynomial algorithms. Since d may be large when the algorithm is called with a long list of denominations (e.g., $c = (1, 2, 3, 4, 5, \ldots, 100)$), we see that BruteForceChange can take a very long time to execute.

1.6 Big-O Notation

Computer scientists use the *Big-O notation* to describe concisely the running time of an algorithm. If we say that the running time of an algorithm is quadratic, or $O(n^2)$, it means that the running time of the algorithm on an input of size n is limited by a quadratic function of n. That limit may be $99.7n^2$ or $0.001n^2$ or $5n^2 + 3.2n + 99993$; the main factor that describes the growth rate of the running time is the term that grows the fastest with respect to n, for example n^2 when compared to terms like $3.2n$, or 99993. All functions with a leading term of n^2 have more or less the same rate of growth, so we lump them into one class which we call $O(n^2)$. The difference in behavior between two quadratic functions in that class, say $99.7n^2$ and $5n^2 + 3.2n + 99993$, is negligible when compared to the difference in behavior between two functions in different classes, say $5n^2 + 3.2n$ and $1.2n^3$. Of course, $99.7n^2$ and $5n^2$ are different functions and we would prefer an algorithm that takes $5n^2$ operations to an algorithm that takes $99.7n^2$. However, computer scientists typically ignore the leading constant and pay attention only to the fastest growing term.

When we write $f(n) = O(n^2)$, we mean that the function $f(n)$ does not grow faster than a function with a leading term of cn^2, for a suitable choice of the constant c. In keeping with the healthy dose of pessimism toward an algorithm's performance, we measure an algorithm's efficiency as its worst case efficiency, which is the largest amount of time an algorithm can take given the worst possible input of a given size. The advantage to considering the worst case efficiency of an algorithm is that we are guaranteed that our algorithm will never behave worse than our worst case estimate, so we are never surprised or disappointed. Thus, when we derive a Big-O bound, it is a bound on the worst case efficiency.

Chapter 2: Algorithm Design Techniques

Over the last half a century, computer scientists have discovered that many algorithms share similar ideas, even though they solve very different problems. There appear to be relatively few basic techniques that can be applied when designing an algorithm, and we cover some of them later in various programming challenges in this book. For now we will mention the most common algorithm design techniques, so that future examples can be categorized in terms of the algorithm's design methodology.

To illustrate the design techniques, we will consider a very simple problem that plagued nearly everyone before the era of mobile phones when people used cordless phones. Suppose your cordless phone rings, but you have misplaced the handset somewhere in your home. How do you find it? To complicate matters, you have just walked into your home with an armful of groceries, and it is dark out, so you cannot rely solely on eyesight.

2.1 Exhaustive Search Algorithms

An *exhaustive search*, or *brute force*, algorithm examines every possible alternative to find one particular solution. For example, if you used the brute force algorithm to find the ringing telephone, you would ignore the ringing of the phone, as if you could not hear it, and simply walk over every square inch of your home checking to see if the phone was present. You probably would not be able to answer the phone before it stopped ringing, unless you were very lucky, but you would be guaranteed to eventually find the phone no matter where it was.

BruteForceChange is a brute force algorithm, and our programming challenges include some additional examples of such algorithms—these are the easiest algorithms to design, and sometimes they work for certain practical problems. In general, though, brute force algorithms are too slow to be practical for anything but the smallest instances and you should always think how to avoid the brute force algorithms or how to finesse them into faster versions.

2.2 Branch-and-Bound Algorithms

In certain cases, as we explore the various alternatives in a brute force algorithm, we discover that we can omit a large number of alternatives, a technique that is often called *branch-and-bound*.

Suppose you were exhaustively searching the first floor and heard the phone ringing above your head. You could immediately rule out the need to search the basement or the first floor. What may have taken three hours may now only take one, depending on the amount of space that you can rule out.

2.3 Greedy Algorithms

Many algorithms are iterative procedures that choose among a number of alternatives at each iteration. For example, a cashier can view the Change Problem as a series of decisions he or she has to make: which coin (among d denominations) to return first, which to return second, and so on. Some of these alternatives may lead to correct solutions while others may not.

Greedy algorithms choose the "most attractive" alternative at each iteration, for example, the largest denomination possible. In the case of the US denominations, CHANGE used quarters, then dimes, then nickels, and finally pennies (in that order) to make change. Of course, we showed that this greedy strategy produced incorrect results when certain new denominations were included.

In the telephone example, the corresponding greedy algorithm would simply be to walk in the direction of the telephone's ringing until you found it. The problem here is that there may be a wall (or a fragile vase) between you and the phone, preventing you from finding it. Unfortunately, these sorts of difficulties frequently occur in most realistic problems. In many cases, a greedy approach will seem "obvious" and natural, but will be subtly wrong.

2.4 Dynamic Programming Algorithms

Some algorithms break a problem into smaller subproblems and use the solutions of the subproblems to construct the solution of the larger one.

During this process, the number of subproblems may become very large, and some algorithms solve the same subproblem repeatedly, needlessly increasing the running time. Dynamic programming organizes computations to avoid recomputing values that you already know, which can often save a great deal of time.

The Ringing Telephone Problem does not lend itself to a dynamic programming solution, so we consider a different problem to illustrate the technique. Suppose that instead of answering the phone you decide to play the "Rocks" game with two piles of rocks, say ten in each. In each turn, one player may take either one rock (from either pile) or two rocks (one from each pile). Once the rocks are taken, they are removed from play. The player that takes the last rock wins the game. You make the first move. We encourage you to play this game using our interactive puzzle.

To find the winning strategy for the 10+10 game, we can construct a table, which we can call R, shown in Figure 2.1. Instead of solving a problem with 10 rocks in each pile, we will solve a more general problem with n rocks in one pile and m rocks in the other pile (the $n+m$ game) where n and m are arbitrary non-negative integers.

If Player 1 can always win the $n+m$ game, then we would say $R(n,m) = W$, but if Player 1 has no winning strategy against a player that always makes the right moves, we would write $R(n,m) = L$. Computing $R(n,m)$ for arbitrary n and m seems difficult, but we can build on smaller values. Some games, notably $R(0,1)$, $R(1,0)$, and $R(1,1)$, are clearly winning propositions for Player 1 since in the first move, Player 1 can win. Thus, we fill in entries $(1,1)$, $(0,1)$, and $(1,0)$ as W. See Figure 2.1(a).

After the entries $(0,1)$, $(1,0)$, and $(1,1)$ are filled, one can try to fill other entries. For example, in the $(2,0)$ case, the only move that Player 1 can make leads to the $(1,0)$ case that, as we already know, is a winning position for his opponent. A similar analysis applies to the $(0,2)$ case, leading to the table in Figure 2.1(b).

In the $(2,1)$ case, Player 1 can make three different moves that lead respectively to the games of $(1,1)$, $(2,0)$, or $(1,0)$. One of these cases, $(2,0)$, leads to a losing position for his opponent and therefore $(2,1)$ is a winning position. The case $(1,2)$ is symmetric to $(2,1)$, so we have the table shown at Figure 2.1(c).

Now we can fill in $R(2,2)$. In the $(2,2)$ case, Player 1 can make three different moves that lead to entries $(2,1)$, $(1,2)$, and $(1,1)$. All of these entries are winning positions for his opponent and therefore $R(2,2) = L$,

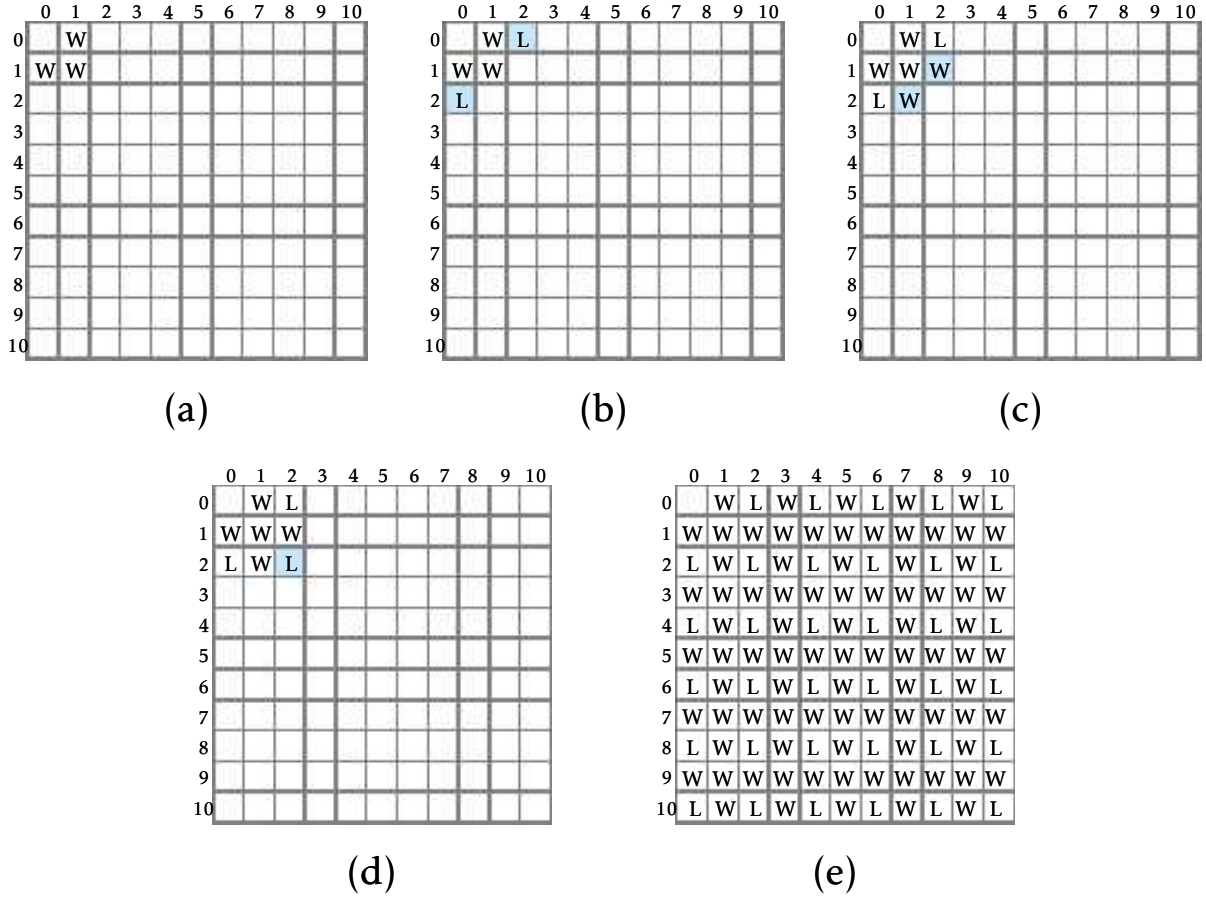


Figure 2.1: Table R for the $10 + 10$ Rocks game.

see Figure 2.1(d).

We can proceed filling in R in this way by noticing that for the entry (i,j) to be L, the entries above, diagonally to the left, and directly to the left, must be W. These entries ($(i-1,j)$, $(i-1,j-1)$, and $(i,j-1)$) correspond to the three possible moves that Player 1 can make. See Figure 2.1(e).

The Rocks algorithm determines if Player 1 wins or loses. If Player 1 wins in an $n+m$ game, Rocks returns W. If Player 1 loses, Rocks returns L. We introduced an artificial initial condition, $R(0,0) = L$ to simplify the pseudocode.

```
Rocks(n, m):
R(0,0) ← L
for i from 1 to n:
    if R(i − 1, 0) = W:
        R(i, 0) ← L
    else:
        R(i, 0) ← W
for j from 1 to m:
    if R(0, j − 1) = W:
        R(0, j) ← L
    else:
        R(0, j) ← W
for i from 1 to n:
    for j from 1 to m:
        if R(i − 1, j − 1) = W and R(i, j − 1) = W and R(i − 1, j) = W:
            R(i, j) ← L
        else:
            R(i, j) ← W
return R(n, m)
```

A faster algorithm to solve the Rocks puzzle relies on the simple pattern in R, and checks if n and m are both even, in which case the player loses (see table above).

```
FastRocks(n, m):
if n and m are both even:
    return L
else:
    return W
```

However, though FastRocks is more efficient than Rocks, it may be difficult to modify it for similar games, for example, a game in which each player can move up to three rocks at a time from the piles. This is one example where the slower algorithm is more instructive than a faster one.

Exercise Break. Play the Three Rocks game using our interactive puzzle and construct the dynamic programming table similar to the table above for this game.

2.5 Recursive Algorithms

Recursion is one of the most ubiquitous algorithmic concepts. Simply, an algorithm is recursive if it calls itself.

The *Towers of Hanoi puzzle* consists of three pegs, which we label from left to right as 1, 2, and 3, and a number of disks of decreasing radius, each with a hole in the center. The disks are initially stacked on the left peg (peg 1) so that smaller disks are on top of larger ones. The game is played by moving one disk at a time between pegs. You are only allowed to place smaller disks on top of larger ones, and any disk may go onto an empty peg. The puzzle is solved when all of the disks have been moved from peg 1 to peg 3. Try our interactive puzzle Hanoi Towers to figure out how to move all disks from one peg to another.

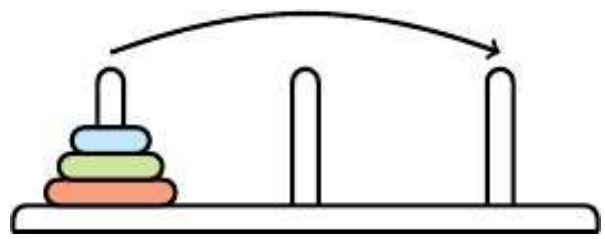
Towers of Hanoi Problem
Output a list of moves that solves the Towers of Hanoi.

Input: An integer n.
Output: A sequence of moves that solve the n-disk Towers of Hanoi puzzle.

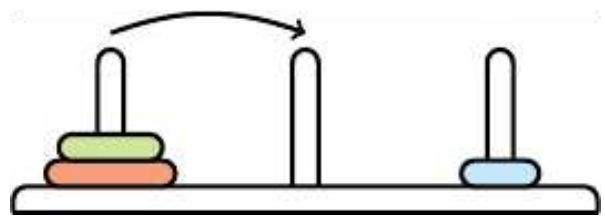
Solving the puzzle with one disk is easy: move the disk to the right peg. The two-disk puzzle is not much harder: move the small disk to the middle peg, then the large disk to the right peg, then the small disk to the right peg to rest on top of the large disk.

The three-disk puzzle is somewhat harder, but the following sequence of seven moves solves it:

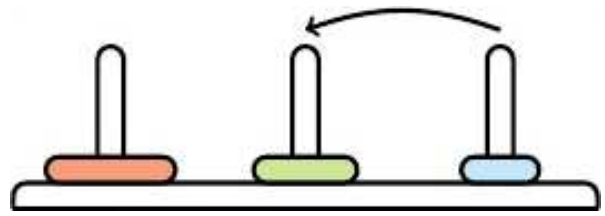
1. Move disk from peg 1 to peg 3



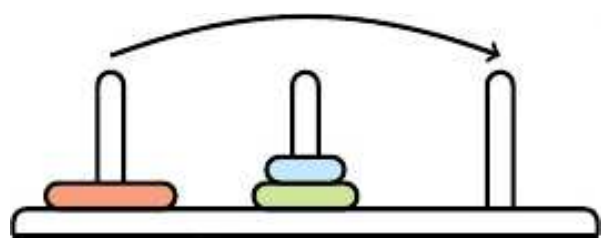
2. Move disk from peg 1 to peg 2



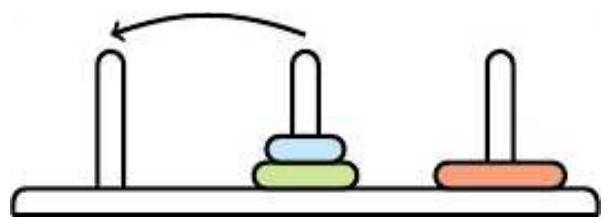
3. Move disk from peg 3 to peg 2



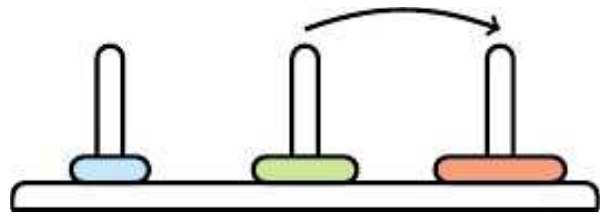
4. Move disk from peg 1 to peg 3



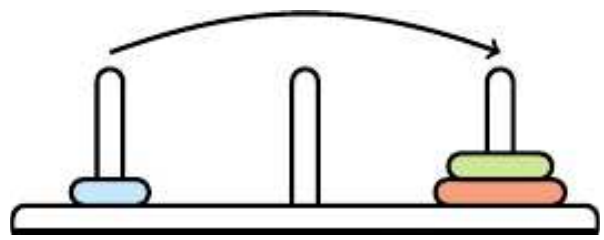
5. Move disk from peg 2 to peg 1



6. Move disk from peg 2 to peg 3



7. Move disk from peg 1 to peg 3



Now we will figure out how many steps are required to solve a four-disk puzzle. You cannot complete this game without moving the largest disk. However, in order to move the largest disk, we first had to move all the smaller disks to an empty peg. If we had four disks instead of three, then we would first have to move the top three to an empty peg (7 moves), then move the largest disk (1 move), then again move the three disks from their temporary peg to rest on top of the largest disk (another 7 moves). The whole procedure will take $7 + 1 + 7 = 15$ moves.

More generally, to move a stack of size n from the left to the right peg, you first need to move a stack of size $n-1$ from the left to the middle peg, and then from the middle peg to the right peg once you have moved the n-th disk to the right peg. To move a stack of size $n-1$ from the middle to the right, you first need to move a stack of size $n-2$ from the middle to the left, then move the $(n-1)$-th disk to the right, and then move the stack of size $n-2$ from the left to the right peg, and so on.

At first glance, the Towers of Hanoi Problem looks difficult. However, the following *recursive algorithm* solves the Towers of Hanoi Problem with just 9 lines!

```
HanoiTowers(n, fromPeg, toPeg)
if n = 1:
   output "Move disk from peg fromPeg to peg toPeg"
   return
unusedPeg ← 6 – fromPeg – toPeg
HanoiTowers(n – 1, fromPeg, unusedPeg)
output "Move disk from peg fromPeg to peg toPeg"
HanoiTowers(n – 1, unusedPeg, toPeg)
return
```

The variables *fromPeg*, *toPeg*, and *unusedPeg* refer to the three different pegs so that HanoiTowers$(n, 1, 3)$ moves n disks from the first peg to the third peg. The variable *unusedPeg* represents which of the three pegs can serve as a temporary destination for the first $n-1$ disks. Note that *fromPeg* + *toPeg* + *unusedPeg* is always equal to $1+2+3=6$, so the value of the variable *unusedPeg* can be computed as $6 -$ *fromPeg* $-$ *toPeg*. Table below shows the result of $6 -$ *fromPeg* $-$ *toPeg* for all possible values of *fromPeg* and *toPeg*.

fromPeg	*toPeg*	*unusedPeg*
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

After computing *unusedPeg* as $6 -$ *fromPeg* $-$ *toPeg*, the statements

```
HanoiTowers(n – 1, fromPeg, unusedPeg)
output "Move disk from peg fromPeg to peg toPeg"
HanoiTowers(n – 1, unusedPeg, toPeg)
return
```

solve the smaller problem of moving the stack of size $n-1$ first to the temporary space, moving the largest disk, and then moving the $n-1$ remaining disks to the final destination. Note that we do not have to specify which disk the player should move from *fromPeg* to *toPeg*: it is always the top disk currently residing on *fromPeg* that gets moved.

Although the Hanoi Tower solution can be expressed in just 9 lines of pseudocode, it requires a surprisingly long time to run. To solve a five-disk tower requires 31 moves, but to solve a hundred-disk tower would require more moves than there are atoms on Earth. The fast growth of the number of moves that HanoiTowers requires is easy to see by noticing that every time HanoiTowers$(n,1,3)$ is called, it calls itself twice for $n-1$, which in turn triggers four calls for $n-2$, and so on.

We can illustrate this situation in a *recursion tree*, which is shown in Figure 2.2. A call to HanoiTowers$(4,1,3)$ results in calls HanoiTowers$(3,1,2)$ and HanoiTowers$(3,2,3)$; each of these results in calls to HanoiTowers$(2,1,3)$, HanoiTowers$(2,3,2)$ and HanoiTowers$(2,2,1)$, HanoiTowers$(2,1,3)$, and so on. Each call to the subroutine HanoiTowers requires some amount of time, so we would like to know how much time the algorithm will take.

To calculate the running time of HanoiTowers of size n, we denote the number of disk moves that HanoiTowers(n) performs as $T(n)$ and notice that the following equation holds:

$$T(n) = 2 \cdot T(n-1) + 1.$$

Starting from $T(1) = 1$, this recurrence relation produces the sequence:

$$1, 3, 7, 15, 31, 63,$$

and so on. We can compute $T(n)$ by adding 1 to both sides and noticing

$$T(n) + 1 = 2 \cdot T(n-1) + 1 + 1 = 2 \cdot (T(n-1) + 1).$$

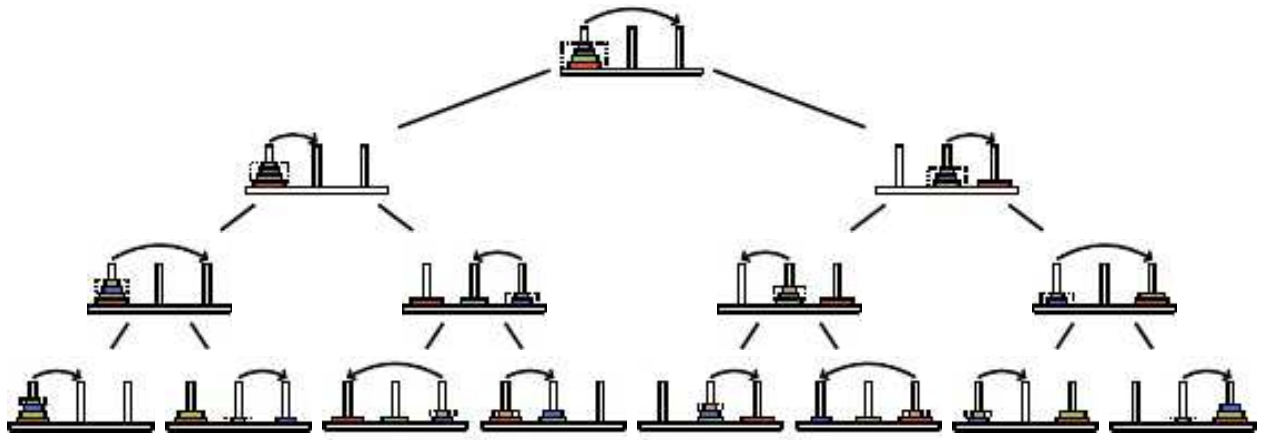
If we introduce a new variable, $U(n) = T(n) + 1$, then $U(n) = 2 \cdot U(n-1)$. Thus, we have changed the problem to the following recurrence relation.

$$U(n) = 2 \cdot U(n-1).$$

Starting from $U(1) = 2$, this gives rise to the sequence

$$2, 4, 8, 16, 32, 64, \ldots$$

implying that at $U(n) = 2^n$ and $T(n) = U(n) - 1 = 2^n - 1$. Thus, HanoiTowers$(n)$ is an exponential algorithm.



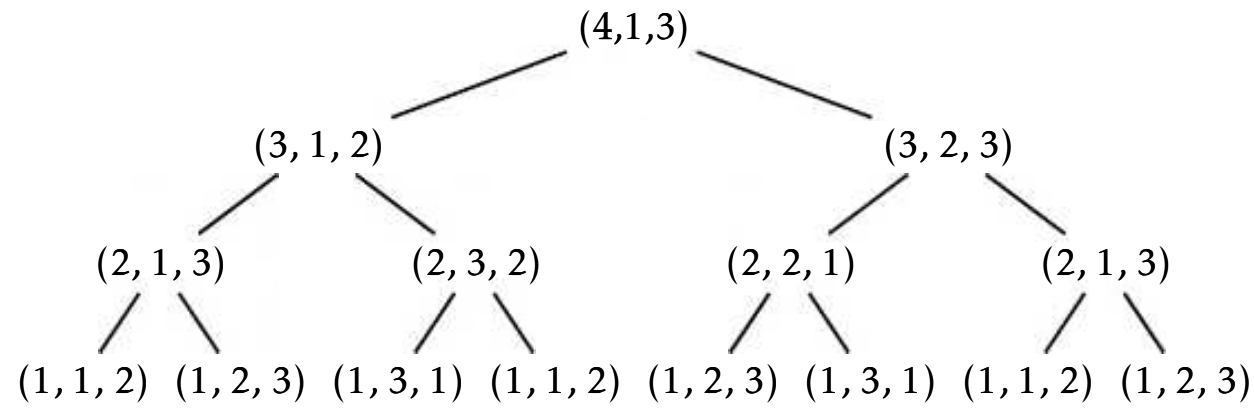


Figure 2.2: The recursion tree for a call to HanoiTowers(4,1,3), which solves the Towers of Hanoi problem of size 4. At each point in the tree, (i,j,k) stands for HanoiTowers(i,j,k).

2.6 Divide-and-Conquer Algorithms

One big problem may be hard to solve, but two problems that are half the size may be significantly easier. In these cases, divide-and-conquer algorithms fare well by doing just that: splitting the problem into smaller subproblems, solving the subproblems independently, and combining the solutions of subproblems into a solution of the original problem. The situation is usually more complicated than this and after splitting one problem into subproblems, a divide-and-conquer algorithm usually splits these subproblems into even smaller sub-subproblems, and so on, until it reaches a point at which it no longer needs to recurse. A critical step in many divide-and-conquer algorithms is the recombining of solutions to subproblems into a solution for a larger problem.

To give an example of a divide-and conquer algorithm, we will consider the sorting problem:

Sorting Problem
Sort a list of integers.

Input: A list of n distinct integers $a = (a_1, a_2, \ldots, a_n)$.
Output: Sorted list of integers, that is, a reordering $b = (b_1, b_2, \ldots, b_n)$ of integers from a such that $b_1 < b_2 < \cdots < b_n$.

SelectionSort is a simple iterative method to solve the Sorting Problem. It first finds the smallest element in a, and moves it to the first position by swapping it with whatever happens to be in the first position (i.e., a_1). Next, it finds the second smallest element in a, and moves it to the second position, again by swapping with a_2 . At the i-th iteration, SelectionSort finds the i-th smallest element in a, and moves it to the i-th position. This is an intuitive approach at sorting, but is not the fastest one.

If $a = (7, 92, 87, 1, 4, 3, 2, 6)$, SelectionSort$(a, 8)$ takes the following seven steps:

$$(7, 92, 87, 1, 4, 3, 2, 6)$$
$$(1, 92, 87, 7, 4, 3, 2, 6)$$
$$(1, 2, 87, 7, 4, 3, 92, 6)$$
$$(1, 2, 3, 7, 4, 87, 92, 6)$$

$(1,2,3,4,7,87,92,6)$
$(1,2,3,4,6,87,92,7)$
$(1,2,3,4,6,7,92,87)$
$(1,2,3,4,6,7,87,92)$

MergeSort is a canonical example of divide-and-conquer sorting algorithm that is much faster than SelectionSort. We begin from the problem of *merging*, in which we want to combine two sorted lists $List_1$ and $List_2$ into a single sorted list.

$List_1$	2 5 7 8 |	2 5 7 8 |	2 5 7 8 |	2 5 7 8 |	2 5 7 8 |	2 5 7 8
$List_2$	3 4 6 |	3 4 6 |	3 4 6 |	3 4 6 |	3 4 6 |	3 4 6
sortedList	2	3	4	5	6	7 8

The Merge algorithm combines two sorted lists into a single sorted list in $O(|List_1| + |List_2|)$ time by iteratively choosing the smallest remaining element in $List_1$ and $List_2$ and moving it to the growing sorted list.

```
Merge(List1, List2):
SortedList ← empty list
while both List1 and List2 are non-empty:
   if the smallest element in List1 is smaller than the smallest element in List2
      move the smallest element from List1 to the end of SortedList
   else:
      move the smallest element from List2 to the end of SortedList
move any remaining elements from either List1 or List2 to the end of SortedList
return SortedList
```

Merge would be useful for sorting an arbitrary list if we knew how to divide an arbitrary (unsorted) list into two already sorted half-sized lists. However, it may seem that we are back to where we started, except now we have to sort two smaller lists instead of one big one. Yet sorting two smaller lists is a preferable algorithmic problem. To see why, let's consider the MergeSort algorithm, which divides an unsorted list into two parts and then recursively conquers each smaller sorting problem before merging the sorted lists.

```
MergeSort(List):
if List consists of a single element:
   return List
FirstHalf ← first half of List
SecondHalf ← second half of List
SortedFirstHalf ← MergeSort(FirstHalf)
SortedSecondHalf ← MergeSort(SecondHalf)
SortedList ← Merge(SortedFirstHalf, SortedSecondHalf)
return SortedList
```

Stop and Think. What is the runtime of MergeSort?

Figure 2.3 shows the recursion tree of MergeSort, consisting of $\log_2 n$ levels, where n is the size of the original unsorted list. At the bottom level, we must merge two sorted lists of approximately $n/2$ elements each, requiring $O(n/2+n/2) = O(n)$ time. At the next highest level, we must merge four lists of $n/4$ elements, requiring $O(n/4+n/4+n/4+n/4) = O(n)$ time. This pattern can be generalized: the i-th level contains 2^i lists, each having approximately $n/2^i$ elements, and requires $O(n)$ time to merge. Since there are $\log_2 n$ levels in the recursion tree, MergeSort requires $O(n \log_2 n)$ runtime overall, which offers a speedup over a naive $O(n^2)$ sorting algorithm.

2.7 Randomized Algorithms

If you happen to have a coin, then before even starting to search for the phone, you could toss it to decide whether you want to start your search on the first floor if the coin comes up heads, or on the second floor if the coin comes up tails. If you also happen to have a die, then after deciding on the second floor of your mansion, you could roll it to decide in which of the six rooms on the second floor to start your search. Although tossing coins and rolling dice may be a fun way to search for the phone, it is certainly not the intuitive thing to do, nor is it at all clear whether it gives you any algorithmic advantage over a deterministic algorithm. Our programming challenges will help you to learn why randomized algorithms are useful and why some of them have a competitive advantage over deterministic algorithms.

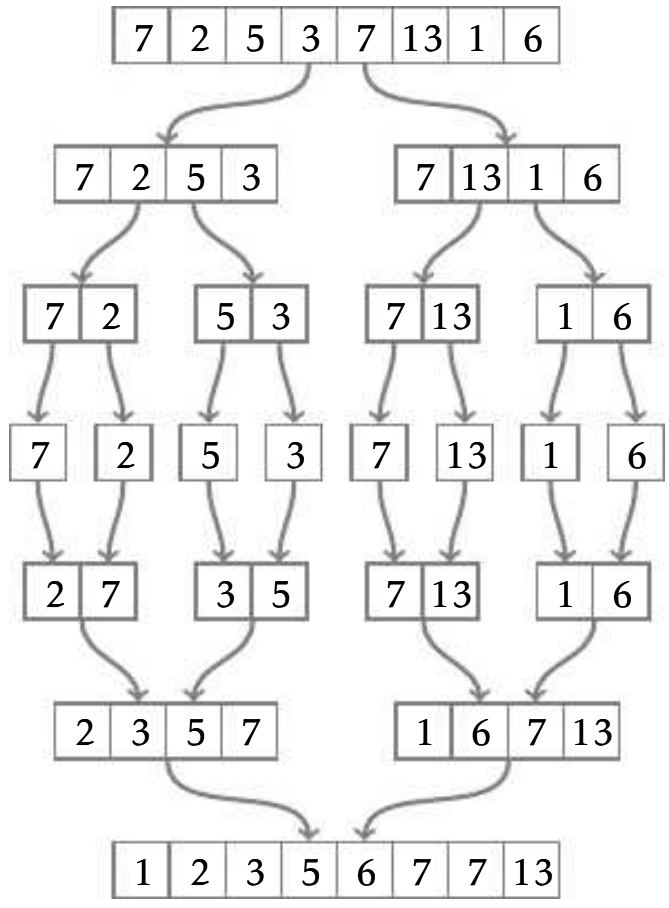


Figure 2.3: The recursion tree for sorting an 8-element array with MergeSort. The divide (upper) steps consist of $\log_2 8 = 3$ levels, where the input array is split into smaller and smaller subarrays. The conquer (lower) steps consist of the same number of levels, as the sorted subarrays are merged back together.

To give an example of a randomized algorithm, we will first discuss a fast sorting technique called QuickSort. It selects an element m (typically, the first) from an array c and simply partitions the array into two subarrays: c_{small}, containing all elements from c that are smaller than m; and c_{large} containing all elements larger than m.

This partitioning can be done in linear time, and by following a divide-and-conquer strategy, QuickSort recursively sorts each subarray in the same way. The sorted list is easily created by simply concatenating the sorted c_{small}, element m, and the sorted c_{large}.

```
QuickSort(c):
if c consists of a single element:
   return c
m ← c[1]
determine the set of elements c_small smaller than m
determine the set of elements c_large larger than m
QuickSort(c_small)
QuickSort(c_large)
combine c_small, m, and c_large into a single sorted array c_sorted
return c_sorted
```

It turns out that the running time of QuickSort depends on how lucky we are with our selection of the element m. If we happen to choose m in such a way that the array c is split into even halves (i.e., $|c_{small}| = |c_{large}|$), then

$$T(n) = 2T\binom{n}{2} + a \cdot n,$$

where $T(n)$ represents the time taken by QuickSort to sort an array of n numbers, and $a \cdot n$ represents the time required to split the array of size n into two parts; a is a positive constant. This is exactly the same recurrence as in MergeSort that leads to $O(n \log n)$ running time.

However, if we choose m in such a way that it splits c unevenly (e.g., an extreme case occurs when c_{small} is empty and c_{large} has $n-1$ elements), then the recurrence looks like

$$T(n) = T(n-1) + a \cdot n.$$

This is the recurrence that leads to $O(n^2)$ running time, something we want to avoid. Indeed, QuickSort takes quadratic time to sort the array

$(n, n-1, \ldots, 2, 1)$. Worse yet, it requires $O(n^2)$ time to process $(1, 2, \ldots, n-1, n)$, which seems unnecessary since the array is already sorted.

The QuickSort algorithm so far seems like a bad imitation of MergeSort. However, if we can choose a good "splitter" m that breaks an array into two equal parts, we might improve the running time. To achieve $O(n \log n)$ running time, it is not actually necessary to find a perfectly equal (50/50) split. For example, a split into approximately equal parts of size, say, 51/49 will also work. In fact, one can prove that the algorithm will achieve $O(n \log n)$ running time as long as the sets c_{small} and c_{large} are both larger in size than $n/4$.

It implies that, of n possible choices for m as elements of the array c, at least $\frac{3n}{4} - \frac{n}{4} = \frac{n}{2}$ of them make good splitters! In other words, if we randomly choose m (i.e., every element of the array c has the same probability to be chosen), there is at least a 50% chance that it will be a good splitter. This observation motivates the following randomized algorithm:

```
RandomizedQuickSort(c):
if c consists of a single element:
   return c
randomly select an element m from c
determine the set of elements c_small smaller than m
determine the set of elements c_large larger than m
RandomizedQuickSort(c_small)
RandomizedQuickSort(c_large)
combine c_small, m, and c_large into a single sorted array c_sorted
return c_sorted
```

RandomizedQuickSort is a fast algorithm in practice, but its worst case running time remains $O(n^2)$ since there is still a possibility that it selects bad splitters. Although the behavior of a randomized algorithm varies on the same input from one execution to the next, one can prove that its *expected* running time is $O(n \log n)$. The running time of a randomized algorithm is a *random variable*, and computer scientists are often interested in the mean value of this random variable which is referred to as the expected running time.

The key advantage of randomized algorithms is performance: for many practical problems randomized algorithms are faster (in the sense of expected running time) than the best known deterministic algorithms.

Another attractive feature of randomized algorithms, as illustrated by RandomizedQuickSort, is their simplicity.

We emphasize that RandomizedQuickSort, despite making random decisions, always returns the correct solution of the sorting problem. The only variable from one run to another is its running time, not the result. In contrast, other randomized algorithms usually produce incorrect (or, more gently, *approximate*) solutions. Randomized algorithms that always return correct answers are called *Las Vegas algorithms*, while algorithms that do not are called *Monte Carlo algorithms*. Of course, computer scientists prefer Las Vegas algorithms to Monte Carlo algorithms, but the former are often difficult to come by.

Chapter 3: Programming Challenges

To introduce you to our automated grading system, we will discuss two simple programming challenges and walk you through a step-by-step process of solving them. We will encounter several common pitfalls and will show you how to fix them.

Below is a brief overview of what it takes to solve a programming challenge in five steps:

Reading problem statement. Problem statement specifies the input-output format, the constraints for the input data as well as time and memory limits. Your goal is to implement a fast program that solves the problem and works within the time and memory limits.

Designing an algorithm. When the problem statement is clear, start designing an algorithm and don't forget to prove that it works correctly.

Implementing an algorithm. After you developed an algorithm, start implementing it in a programming language of your choice.

Testing and debugging your program. Testing is the art of revealing bugs. Debugging is the art of exterminating the bugs. When your program is ready, start testing it! If a bug is found, fix it and test again.

Submitting your program to the grading system. After testing and debugging your program, submit it to the grading system and wait for the message "Good job!". In the case you see a different message, return back to the previous stage.

3.1 Sum of Two Digits

Sum of Two Digits Problem
Compute the sum of two single digit numbers.

Input: Two single digit numbers.
Output: The sum of these numbers.

$2 + 3 = 5$

We start from this ridiculously simple problem to show you the pipeline of reading the problem statement, designing an algorithm, implementing it, testing and debugging your program, and submitting it to the grading system.

Input format. Integers a and b on the same line (separated by a space).

Output format. The sum of a and b.

Constraints. $0 \le a, b \le 9$.

Sample.

Input:

```
9 7
```

Output:

```
16
```

Time limits (sec.):

C	C++	Java	Python	C#	Haskell	JavaScript	Ruby	Rust	Scala
1	1	1.5	5	1.5	2	5	5	1	3

Memory limit. 512 Mb.

For this trivial problem, we will skip "Designing an algorithm" step and will move right to the pseudocode.

```
SumOfTwoDigits(a, b):
return a + b
```

Since the pseudocode does not specify how we input a and b, below we provide solutions in C++, Java, and Python3 programming languages as well as recommendations on compiling and running them. You can copy-and-paste the code to a file, compile/run it, test it on a few datasets, and then submit (the source file, not the compiled executable) to the grading system. Needless to say, we assume that you know the basics of one of programming languages that we use in our grading system.

C++

```
#include <iostream>

int main() {
  int a = 0;
  int b = 0;
  std::cin >> a;
  std::cin >> b;
  std::cout << a + b;
  return 0;
}
```

Save this to a file (say, `aplusb.cpp`), compile it, run the resulting executable, and enter two numbers (on the same line).

Java

```
import java.util.Scanner;

class APlusB {
  public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int a = s.nextInt();
    int b = s.nextInt();
    System.out.println(a + b);
  }
}
```

Save this to a file `APlusB.java`, compile it, run the resulting executable, and enter two numbers (on the same line).

Python3

```
# Uses python3
import sys

input = sys.stdin.read()
tokens = input.split()
a = int(tokens[0])
b = int(tokens[1])
print(a + b)
```

Save this to a file (say, `aplusb.py`), run it, and enter two numbers on the same line. To indicate the end of input, press `ctrl-d/ctrl-z`. (The first line in the code above tells the grading system to use Python3 rather Python2.)

Your goal is to implement an algorithm that produces a correct result under the given time and memory limits for any input satisfying the given constraints. You do not need to check that the input data satisfies the constraints, e.g., for the Sum of Two Digits Problem you do not need to check that the given integers a and b are indeed single digit integers (this is guaranteed).

3.2 Maximum Pairwise Product

Maximum Pairwise Product Problem
Find the maximum product of two distinct numbers in a sequence of non-negative integers.

Input: A sequence of non-negative integers.
Output: The maximum value that can be obtained by multiplying two different elements from the sequence.

	5	6	2	7	4
5		30	10	35	20
6	30		12	42	24
2	10	12		7	4
7	35	42	14		28
4	20	24	8	28	

Given a sequence of non-negative integers $a_1, \ldots, a_n$, compute

$$\max_{1 \le i \ne j \le n} a_i \cdot a_j .$$

Note that i and j should be different, though it may be the case that $a_i = a_j$.

Input format. The first line contains an integer n. The next line contains n non-negative integers $a_1, \ldots, a_n$ (separated by spaces).

Output format. The maximum pairwise product.

Constraints. $2 \le n \le 2 \cdot 10^5$; $0 \le a_1, \ldots, a_n \le 2 \cdot 10^5$.

Sample 1.

Input:

```
3
1 2 3
```

Output:

```
6
```

Sample 2.

Input:

```
10
7 5 14 2 8 8 10 1 2 3
```

Output:

```
140
```

Time and memory limits. The same as for the previous problem.

3.2.1 Naive Algorithm

A naive way to solve the Maximum Pairwise Product Problem is to go through all possible pairs of the input elements $A[1 \dots n] = [a_1, \dots, a_n]$ and to find a pair of distinct elements with the largest product:

```
MaxPairwiseProductNaive(A[1 ... n]):
product ← 0
for i from 1 to n:
    for j from 1 to n:
        if i ≠ j:
            if product < A[i] · A[j]:
                product ← A[i] · A[j]
return product
```

This code can be optimized and made more compact as follows.

```
MaxPairwiseProductNaive(A[1 ... n]):
product ← 0
for i from 1 to n:
    for j from i + 1 to n:
        product ← max(product, A[i] · A[j])
return product
```

Implement this algorithm in your favorite programming language. If you are using C++, Java, or Python3, you may want to download the starter files (we provide starter solutions in these three languages for all the problems in the book). For other languages, you need to implement your solution from scratch.

Starter solutions for C++, Java, and Python3 are shown below.

C++

```
#include <iostream>
#include <vector>

using std::vector;
using std::cin;
using std::cout;
using std::max;

int MaxPairwiseProduct(const vector<int>& numbers) {
  int product = 0;
  int n = numbers.size();
  for (int i = 0; i < n; ++i) {
    for (int j = i + 1; j < n; ++j) {
      product = max(product, numbers[i] * numbers[j]);
    }
  }
  return product;
}

int main() {
    int n;
    cin >> n;
    vector<int> numbers(n);
    for (int i = 0; i < n; ++i) {
        cin >> numbers[i];
    }

    int product = MaxPairwiseProduct(numbers);
    cout << product << "\n";
    return 0;
}
```

Java

```
import java.util.*;
import java.io.*;
```

```
public class MaxPairwiseProduct {
    static int getMaxPairwiseProduct(int[] numbers) {
        int product = 0;
        int n = numbers.length;
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                product = Math.max(product,
                                   numbers[i] * numbers[j]);
            }
        }
        return product;
    }

    public static void main(String[] args) {
        FastScanner scanner = new FastScanner(System.in);
        int n = scanner.nextInt();
        int[] numbers = new int[n];
        for (int i = 0; i < n; i++) {
            numbers[i] = scanner.nextInt();
        }
        System.out.println(getMaxPairwiseProduct(numbers));
    }

    static class FastScanner {
        BufferedReader br;
        StringTokenizer st;

        FastScanner(InputStream stream) {
            try {
                br = new BufferedReader(new
                    InputStreamReader(stream));
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        String next() {
            while (st == null || !st.hasMoreTokens()) {
                try {
```

```
                st = new StringTokenizer(br.readLine());
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return st.nextToken();
    }

    int nextInt() {
        return Integer.parseInt(next());
    }
  }

}
```

Python

```
# Uses python3
n = int(input())
a = [int(x) for x in input().split()]

product = 0

for i in range(n):
    for j in range(i + 1, n):
        product = max(product, a[i] * a[j])

print(product)
```

After submitting this solution to the grading system, many students are surprised when they see the following message:

```
Failed case #4/17: time limit exceeded
```

After you submit your program, we test it on dozens of carefully designed test cases to make sure the program is fast and error proof. As the result, we usually know what kind of errors you made. The message above tells that the submitted program exceeds the time limit on the 4th out of 17 test cases.

Stop and Think. Why the solution does not fit into the time limit?

MaxPairwiseProductNaive performs of the order of n^2 steps on a sequence of length n. For the maximal possible value $n = 2 \cdot 10^5$, the number of steps is of the order $4 \cdot 10^{10}$. Since many modern computers perform roughly 10^8–10^9 basic operations per second (this depends on a machine, of course), it may take tens of seconds to execute MaxPairwiseProductNaive, exceeding the time limit for the Maximum Pairwise Product Problem.

We need a faster algorithm!

3.2.2 Fast Algorithm

In search of a faster algorithm, you play with small examples like $[5, 6, 2, 7, 4]$. Eureka—it suffices to multiply the two largest elements of the array—7 and 6!

Since we need to find the largest and the second largest elements, we need only two scans of the sequence. During the first scan, we find the largest element. During the second scan, we find the largest element among the remaining ones by skipping the element found at the previous scan.

```
MaxPairwiseProductFast(A[1...n]):
index_1 ← 1
for i from 2 to n:
    if A[i] > A[index_1]:
        index_1 ← i
index_2 ← 1
for i from 2 to n:
    if A[i] ≠ A[index_1] and A[i] > A[index_2]:
        index_2 ← i
return A[index_1] · A[index_2]
```

3.2.3 Testing and Debugging

Implement this algorithm and test it using an input $A = [1, 2]$. It will output 2, as expected. Then, check the input $A = [2, 1]$. Surprisingly, it outputs 4. By inspecting the code, you find out that after the first loop, $index_1 = 1$. The algorithm then initializes $index_2$ to 1 and $index_2$ is never

updated by the second for loop. As a result, $index_1 = index_2$ before the return statement. To ensure that this does not happen, you modify the pseudocode as follows:

```
MaxPairwiseProductFast(A[1...n]):
index_1 ← 1
for i from 2 to n:
    if A[i] > A[index_1]:
        index_1 ← i
if index_1 = 1:
    index_2 ← 2
else:
    index_2 ← 1
for i from 1 to n:
    if A[i] ≠ A[index_1] and A[i] > A[index_2]:
        index_2 ← i
return A[index_1] · A[index_2]
```

Check this code on a small datasets $[7, 4, 5, 6]$ to ensure that it produces correct results. Then try an input

```
2
100000  90000
```

You may find out that the program outputs something like 410065408 or even a negative number instead of the correct result 9 000000 000. If it does, this is most probably caused by an *integer overflow*. For example, in C++ programming language a large number like 9000000000 does not fit into the standard `int` type that on most modern machines occupies 4 bytes and ranges from -2^{31} to $2^{31} - 1$, where

$$2^{31} = 2147483648.$$

Hence, instead of using the C++ `int` type you need to use the `int64_t` type when computing the product and storing the result. This will prevent integer overflow as the `int64_t` type occupies 8 bytes and ranges from -2^{63} to $2^{63} - 1$, where

$$2^{63} = 9223372036854775808.$$

You then proceed to testing your program on large data sets, e.g., an array $A[1 \dots 2 \cdot 10^5]$, where $A[i] = i$ for all $1 \le i \le 2 \cdot 10^5$. There are two ways of doing this.

1. Create this array in your program and pass it to MaxPairwiseProductFast (instead of reading it from the standard input).

2. Create a separate program, that writes such an array to a file `dataset.txt`. Then pass this dataset to your program from console as follows:

```
yourprogram < dataset.txt
```

Check that your program processes this dataset within time limit and returns the correct result: 39999800000. You are now confident that the program finally works!

However, after submitting it to the testing system, it fails again...

```
Failed case #5/17: wrong answer
```

But how would you generate a test case that make your program fail and help you to figure out what went wrong?

3.2.4 Can You Tell Me What Error Have I Made?

You are probably wondering why we did not provide you with the 5th out of 17 test datasets that brought down your program. The reason is that nobody will provide you with the test cases in real life!

Since even experienced programmers often make subtle mistakes solving algorithmic problems, it is important to learn how to catch bugs as early as possible. When the authors of this book started to program, they naively thought that nearly all their programs are correct. By now, we know that our programs are *almost never* correct when we first run them.

When you are confident that your program works, you often test it on just a few test cases, and if the answers look reasonable, you consider your work done. However, this is a recipe for a disaster. To make your program *always* work, you should test it on a set of carefully designed test cases. Learning how to implement algorithms as well as test and debug your programs will be invaluable in your future work as a programmer.

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:

```
StressTest(N, M):
while true:
  n ← random integer between 2 and N
  allocate array A[1...n]
  for i from 1 to n:
    A[i] ← random integer between 0 and M
  print(A[1...n])
  result_1 ← MaxPairwiseProductNaive(A)
  result_2 ← MaxPairwiseProductFast(A)
  if result_1 = result_2:
    print("OK")
  else:
    print("Wrong answer: ", result_1, result_2)
    return
```

The `while` loop above starts with generating the length of the input sequence n, a random number between 2 and N. It is at least 2, because the problem statement specifies that $n \geq 2$. The parameter N should be small enough to allow us to explore many tests despite the fact that one of our solutions is slow.

After generating n, we generate an array A with n random numbers from 0 to M and output it so that in the process of the infinite loop we always know what is the current test; this will make it easier to catch an error in the test generation code. We then call two algorithms on A and compare the results. If the results are different, we print them and halt. Otherwise, we continue the while loop.

Let's run StressTest(10, 100 000) and keep our fingers crossed in a hope that it outputs "Wrong answer." We see something like this (the result can be different on your computer because of a different random number generator).

```
...
OK
67232 68874 69499
OK
6132 56210 45236 95361 68380 16906 80495 95298
OK
62180 1856 89047 14251 8362 34171 93584 87362 83341 8784
OK
```

```
21468 16859 82178 70496 82939 44491
OK
68165 87637 74297 2904 32873 86010 87637 66131 82858 82935
Wrong answer: 7680243769 7537658370
```

Hurrah! We've found a test case where MaxPairwiseProductNaive and MaxPairwiseProductFast produce different results, so now we can check what went wrong. Then we can debug this solution on this test case, find a bug, fix it, and repeat the stress test again.

Stop and Think. Do you see anything suspicious in the found dataset?

Note that generating tests automatically and running stress test is easy, but debugging is hard. Before diving into debugging, let's try to generate a smaller test case to simplify it. To do that, we change N from 10 to 5 and M from 100000 to 9.

Stop and Think. Why did we first run StressTest with large parameters N and M and now intend to run it with small N and M?

We then run the stress test again and it produces the following.

```
...
7 3 6
OK
2 9 3 1 9
Wrong answer: 81 27
```

The slow MaxPairwiseProductNaive gives the correct answer 81 ($9 \cdot 9 = 81$), but the fast MaxPairwiseProductFast gives an incorrect answer 27.

Stop and Think. How MaxPairwiseProductFast can possibly return 27?

To debug our fast solution, let's check which two numbers it identifies as two largest ones. For this, we add the following line before the `return` statement of the MaxPairwiseProductFast function:

```
print(index1, index2)
```

After running the stress test again, we see the following.

```
...
7 3 6
1 3
```

```
OK
5
2 9 3 1 9
2 3
Wrong answer: 81 27
```

Note that our solutions worked and then failed on exactly the same test cases as on the previous run of the stress test, because we didn't change anything in the test generator. The numbers it uses to generate tests are pseudorandom rather than random—it means that the sequence looks random, but it is the same each time we run this program. It is a convenient and important property, and you should try to have your programs exhibit such behavior, because deterministic programs (that always give the same result for the same input) are easier to debug than non-deterministic ones.

Now let's examine $index_1 = 2$ and $index_2 = 3$. If we look at the code for determining the second maximum, we will notice a subtle bug. When we implemented a condition on i (such that it is not the same as the previous maximum) instead of comparing i and $index_1$, we compared $A[i]$ with $A[index_1]$. This ensures that the second maximum differs from the first maximum by the value rather than by the index of the element that we select for solving the Maximum Pairwise Product Problem. So, our solution fails on any test case where the largest number is equal to the second largest number. We now change the condition from

$$A[i] \neq A[index_1]$$

to

$$i \neq index_1$$

After running the stress test again, we see a barrage of "OK" messages on the screen. We wait for a minute until we get bored and then decide that MaxPairwiseProductFast is finally correct!

However, you shouldn't stop here, since you have only generated very small tests with $N = 5$ and $M = 10$. We should check whether our program works for larger n and larger elements of the array. So, we change N to 1000 (for larger N, the naive solution will be pretty slow, because its running time is quadratic). We also change M to 200000 and run. We again see the screen filling with words "OK", wait for a minute, and then

decide that (finally!) MaxPairwiseProductFast is correct. Afterwards, we submit the resulting solution to the grading system and pass the Maximum Pairwise Product Problem test!

As you see, even for such a simple problems like Maximum Pairwise Product, it is easy to make subtle mistakes when designing and implementing an algorithm. The pseudocode below presents a more "reliable" way of implementing the algorithm.

```
MaxPairwiseProductFast(A[1...n]):
index ← 1
for i from 2 to n:
  if A[i] > A[index]:
    index ← i
swap A[index] and A[n]
index ← 1
for i from 2 to n – 1:
  if A[i] > A[index]:
    index ← i
swap A[index] and A[n – 1]
return A[n – 1] · A[n]
```

In this book, besides learning how to design and analyze algorithms, you will learn how to implement algorithms in a way that minimizes the chances of making a mistake, and how to test your implementations.

3.2.6 Even Faster Algorithm

The MaxPairwiseProductFast algorithm finds the largest and the second largest elements in about $2n$ comparisons.

Exercise Break. Find two largest elements in an array in $1.5n$ comparisons.

After solving this problem, try the next, even more challenging Exercise Break.

Exercise Break. Find two largest elements in an array in $n + \lceil \log_2 n \rceil - 2$ comparisons.

And if you feel that the previous Exercise Break was easy, here are the next two challenges that you may face at your next interview!

Exercise Break. Prove that no algorithm for finding two largest elements in an array can do this in less than $n + \lceil \log_2 n \rceil - 2$ comparisons.

Exercise Break. What is the fastest algorithm for finding three largest elements?

3.2.7 A More Compact Algorithm

The Maximum Pairwise Product Problem can be solved by the following compact algorithm that uses sorting (in non-decreasing order).

```
MaxPairwiseProductBySorting(A[1...n]):
Sort(A)
return A[n-1]·A[n]
```

This algorithm does more than we actually need: instead of finding two largest elements, it sorts the entire array. For this reason, its running time is $O(n \log n)$, but not $O(n)$. Still, for the given constraints ($2 \le n \le 2 \cdot 10^5$) this is usually sufficiently fast to fit into a second and pass our grader.

3.3 Solving a Programming Challenge in Five Easy Steps

Below we summarize what we've learned in this chapter.

3.3.1 Reading Problem Statement

Start by reading the problem statement that contains the description of a computational task, time and memory limits, and a few sample tests. Make sure you understand how an output matches an input in each sample case.

If time and memory limits are not specified explicitly in the problem statement, the following default values are used.

Time limits (sec.):

C	C++	Java	Python	C#	Haskell	JavaScript	Ruby	Rust	Scala
1	1	1.5	5	1.5	2	5	5	1	3

Memory limit: 512 Mb.

3.3.2 Designing an Algorithm

After designing an algorithm, prove that it is correct and try to estimate its expected running time on the most complex inputs specified in the constraints section. If you laptop performs roughly 10^8–10^9 operations per second, and the maximum size of a dataset in the problem description is $n = 10^5$, then an algorithm with quadratic running time is unlikely to fit into the time limit (since $n^2 = 10^{10}$), while a solution with running time $O(n \log n)$ will. However, an $O(n^2)$ solution will fit if $n = 1000$, and if $n = 100$, even an $O(n^3)$ solutions will fit. Although polynomial algorithms remain unknown for some hard problems in this book, a solution with $O(2^n n^2)$ running time will probably fit into the time limit as long as n is smaller than 20.

3.3.3 Implementing an Algorithm

Start implementing your algorithm in one of the following programming languages supported by our automated grading system: `C`, `C++`, `C#`, `Haskell`, `Java`, `JavaScript`, `Python2`, `Python3`, `Ruby`, or `Scala`. For all problems, we provide starter solutions for `C++`, `Java`, and `Python3`. For other programming languages, you need to implement a solution from scratch. The grading system detects the programming language of your submission automatically, based on the extension of the submission file.

We have reference solutions in `C++`, `Java`, and `Python3` (that we don't share with you) which solve the problem correctly under the given constraints, and spend at most 1/3 of the time limit and at most 1/2 of the memory limit. You can also use other languages, and we've estimated the time limit multipliers for them. However, we have no guarantee that a correct solution for a particular problem running under the given time and memory constraints exists in any of those other languages.

In the Appendix, we list compiler versions and flags used by the grading system. We recommend using the same compiler flags when you test your solution locally. This will increase the chances that your program behaves in the same way on your machine and on the testing machine (note that a buggy program may behave differently when compiled by different compilers, or even by the same compiler with different flags).

3.3.4 Testing and Debugging

Submitting your implementation to the grading system without testing it first is a bad idea! Start with small datasets and make sure that your program produces correct results on all sample datasets. Then proceed to checking how long it takes to process a large dataset. To estimate the running time, it makes sense to implement your algorithm as a function like `solve(dataset)` and then implement an additional procedure `generate()` that produces a large dataset. For example, if an input to a problem is a sequence of integers of length $1 \le n \le 10^5$, then generate a sequence of length 10^5, pass it to your `solve()` function, and ensure that the program outputs the result quickly.

Check the boundary values to ensure that your program processes correctly both short sequences (e.g., with 2 elements) and long sequences (e.g., with 10^5 elements). If a sequence of integers from 0 to, let's say, 10^6 is given as an input, check how your program behaves when it is given a sequence $0, 0, \ldots, 0$ or a sequence $10^6, 10^6, \ldots, 10^6$. Afterwards, check it also on randomly generated data. Check degenerate cases like an empty set, three points on a single line, a tree which consists of a single path of nodes, etc.

After it appears that your program works on all these tests, proceed to stress testing. Implement a slow, but simple and correct algorithm and check that two programs produce the same result (note however that this is not applicable to problems where the output is not unique). Generate random test cases as well as biased tests cases such as those with only small numbers or a small range of large numbers, strings containing a single letter "a" or only two different letters (as opposed to strings composed of all possible Latin letters), and so on. Think about other possible tests which could be peculiar in some sense. For example, if you are generating graphs, try generating trees, disconnected graphs, complete graphs, bipartite graphs, etc. If you generate trees, try generating paths, binary trees,