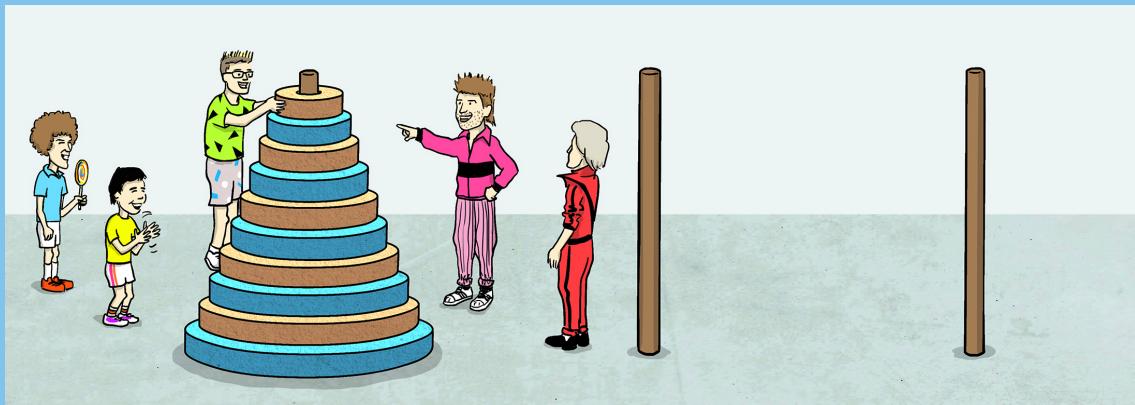


# Learn Algorithms through Programming and Puzzle Solving



Alexander Kulikov and Pavel Pevzner

# Welcome!

Thank you for joining us! This book powers our popular Data Structures and Algorithms online specialization on [Coursera](#) and online MicroMasters program at [edX](#) taken by over 700 000 students. See [the book website](#) for more information about these online courses. 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.

**Stop and Think** questions invite you to slow down and check your knowledge before moving to the next topic.

**Exercise Breaks** offer “just in time” assessments testing your understanding of a topic before moving to the next one.

**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. We encourage you to try our puzzles before attempting to solve the programming challenges.

**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, Kotlin, Ruby, and Rust.

# Contents

<b>About This Book</b>	<b>7</b>
Programming Challenges . . . . .	10
Interactive Algorithmic Puzzles . . . . .	17
What Lies Ahead . . . . .	20
Meet the Authors . . . . .	21
Meet Our Online Co-Instructors . . . . .	22
Acknowledgments . . . . .	23
<b>1 Algorithms and Complexity</b>	<b>25</b>
1.1 What Is an Algorithm? . . . . .	25
1.2 Pseudocode . . . . .	25
1.3 Problem Versus Problem Instance . . . . .	25
1.4 Correct Versus Incorrect Algorithms . . . . .	27
1.5 Fast Versus Slow Algorithms . . . . .	29
1.6 Big-O Notation . . . . .	31
1.6.1 Advantages and Disadvantages . . . . .	35
1.6.2 Five Common Rules . . . . .	36
1.6.3 Visualizing Common Rules . . . . .	37
1.6.4 Frequently Arising Running Times . . . . .	41
<b>2 Algorithm Design Techniques</b>	<b>43</b>
2.1 Exhaustive Search Algorithms . . . . .	43
2.2 Branch-and-Bound Algorithms . . . . .	44
2.3 Greedy Algorithms . . . . .	44
2.4 Dynamic Programming Algorithms . . . . .	47
2.5 Recursive Algorithms . . . . .	50
2.6 Divide-and-Conquer Algorithms . . . . .	56
2.7 Randomized Algorithms . . . . .	60
<b>3 Programming Challenges</b>	<b>65</b>
3.1 Sum of Two Digits . . . . .	66
3.2 Maximum Pairwise Product . . . . .	69
3.2.1 Naive Algorithm . . . . .	70
3.2.2 Fast Algorithm . . . . .	74
3.2.3 Testing and Debugging . . . . .	75

3.2.4	Can You Tell Me What Error Have I Made? . . . . .	77
3.2.5	Stress Testing . . . . .	77
3.2.6	Even Faster Algorithm . . . . .	82
3.2.7	A More Compact Algorithm . . . . .	82
3.3	Solving a Programming Challenge in Five Easy Steps . . . . .	82
3.3.1	Reading Problem Statement . . . . .	83
3.3.2	Designing an Algorithm . . . . .	83
3.3.3	Implementing an Algorithm . . . . .	83
3.3.4	Testing and Debugging . . . . .	84
3.3.5	Submitting to the Grading System . . . . .	85
<b>4</b>	<b>Algorithmic Warm Up</b>	<b>87</b>
4.1	Programming Challenges . . . . .	89
4.1.1	Fibonacci Number . . . . .	89
4.1.2	Last Digit of Fibonacci Number . . . . .	95
4.1.3	Huge Fibonacci Number . . . . .	98
4.1.4	Last Digit of the Sum of Fibonacci Numbers . . . . .	105
4.1.5	Last Digit of the Partial Sum of Fibonacci Numbers .	109
4.1.6	Last Digit of the Sum of Squares of Fibonacci Numbers	111
4.1.7	Greatest Common Divisor . . . . .	114
4.1.8	Least Common Multiple . . . . .	118
4.2	Summary of Algorithmic Ideas . . . . .	120
<b>5</b>	<b>Greedy Algorithms</b>	<b>121</b>
5.1	The Main Idea . . . . .	124
5.1.1	Examples . . . . .	124
5.1.2	Proving Correctness of Greedy Algorithms . . . . .	125
5.1.3	Implementation . . . . .	126
5.2	Programming Challenges . . . . .	127
5.2.1	Money Change . . . . .	127
5.2.2	Maximum Value of the Loot . . . . .	131
5.2.3	Car Fueling . . . . .	136
5.2.4	Maximum Advertisement Revenue . . . . .	141
5.2.5	Collecting Signatures . . . . .	146
5.2.6	Maximum Number of Prizes . . . . .	152
5.2.7	Maximum Salary . . . . .	155

<b>6 Divide-and-Conquer</b>	<b>159</b>
6.1 The Main Idea . . . . .	161
6.1.1 Guess a Number . . . . .	161
6.1.2 Searching Sorted Data . . . . .	165
6.1.3 Finding a White-Black Pair . . . . .	167
6.1.4 Finding a Peak . . . . .	169
6.1.5 Multiplying Integers . . . . .	170
6.1.6 The Master Theorem . . . . .	174
6.2 Programming Challenges . . . . .	181
6.2.1 Binary Search . . . . .	181
6.2.2 Binary Search with Duplicates . . . . .	185
6.2.3 Majority Element . . . . .	190
6.2.4 Speeding-up RANDOMIZEDQUICKSORT . . . . .	194
6.2.5 Number of Inversions . . . . .	197
6.2.6 Organizing a Lottery . . . . .	203
6.2.7 Closest Points . . . . .	209
<b>7 Dynamic Programming</b>	<b>215</b>
7.1 The Main Idea . . . . .	218
7.1.1 Number of Paths . . . . .	218
7.1.2 Dynamic Programming . . . . .	220
7.1.3 Shortest Path in Directed Acyclic Graph . . . . .	221
7.2 Programming Challenges . . . . .	223
7.2.1 Money Change Again . . . . .	224
7.2.2 Primitive Calculator . . . . .	229
7.2.3 Edit Distance . . . . .	234
7.2.4 Longest Common Subsequence of Two Sequences . .	244
7.2.5 Longest Common Subsequence of Three Sequences .	248
7.2.6 Maximum Amount of Gold . . . . .	251
7.2.7 Splitting the Pirate Loot . . . . .	260
7.2.8 Maximum Value of an Arithmetic Expression . . . .	264
7.3 Designing Dynamic Programming Algorithms . . . . .	271
<b>8 Best Programming Practices (Optional)</b>	<b>273</b>
8.1 Language Independent . . . . .	273
8.1.1 Code Format . . . . .	273
8.1.2 Code Structure . . . . .	273
8.1.3 Names and Comments . . . . .	275

8.1.4	Debugging . . . . .	276
8.1.5	Integers and Floating Point Numbers . . . . .	278
8.1.6	Strings . . . . .	279
8.1.7	Ranges . . . . .	280
8.2	C++ Specific . . . . .	282
8.2.1	Code Format . . . . .	282
8.2.2	Code Structure . . . . .	282
8.2.3	Types and Constants . . . . .	284
8.2.4	Classes . . . . .	286
8.2.5	Containers . . . . .	288
8.2.6	Integers and Floating Point Numbers . . . . .	289
8.3	Python Specific . . . . .	290
8.3.1	General . . . . .	290
8.3.2	Code Structure . . . . .	293
8.3.3	Functions . . . . .	295
8.3.4	Strings . . . . .	296
8.3.5	Classes . . . . .	297
8.3.6	Exceptions . . . . .	298
<b>Appendix</b>		<b>301</b>
Compiler Flags . . . . .		301
Frequently Asked Questions . . . . .		302

# 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*

There are many excellent books on Algorithms — why in the world would we 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. As the famous quote by the legendary computer scientist Donald Knuth hints, learning algorithms without implementing them is not unlike learning surgery based solely on reading an anatomy book.

**Intelligent Tutoring System for learning algorithms.** Our goal is to develop an *Intelligent Tutoring System* for learning algorithms 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 and a [microMasters program](#) on edX platform. Since the launch of our MOOCs in 2016, over 700 000 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!

**Like Donald Knuth, learn algorithms by programming them!** Some professors have been concerned 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 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 breakdowns with us.

**Repository of algorithmic coding challenges.** When a student experiences a learning breakdown, s/he needs immediate help in order to proceed. Traditional textbooks do not provide such help, but our automated grading system does! Algorithms is a unique discipline since 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 and helping students to prepare for the coding interviews. 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.

**Solving programming challenges.** When you face a programming challenge, your goal is to implement an 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 give our programming challenges during the interviews.

Your implementation will be automatically checked against many care-

fully selected tests to verify that it always produces a correct answer and fits into the time and memory constraints. 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.

It took us thousands hours to develop the automated homework checking 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. Afterward, 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 this Specialization. This MOOC book discusses 30 of these programming challenges that are included in the first “Algorithmic Toolbox” MOOC.

We are grateful to many students for the feedback that helped us to improve this MOOC that is now ranked among the [top ten computer science courses on Coursera](#) and among [250 best MOOCs of all time](#) across all disciplines and all platforms.

Thank you for joining us!

# Programming Challenges

This book introduces basic algorithmic techniques using the following programming challenges.

$$2 + 3 = 5$$

## Sum of Two Digits

*Compute the sum of two single digit numbers.*

**Input.** Two single digit numbers.

**Output.** The sum of these numbers.

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

## Maximum Pairwise Product

*Find the maximum product of two distinct numbers in a sequence of non-negative integers.*

**Input.** An integer  $n$  and a sequence of  $n$  non-negative integers.

**Output.** The maximum value that can be obtained by multiplying two different elements from the sequence.

$$F_n = F_{n-1} + F_{n-2}$$

## Fibonacci Number

*Compute the  $n$ -th Fibonacci number.*

**Input.** An integer  $n$ .

**Output.**  $n$ -th Fibonacci number.

$$F_{100} = 354\,224\,848\,179$$

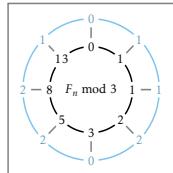
261\,915\,075

## Last Digit of Fibonacci Number

*Compute the last digit of the  $n$ -th Fibonacci number.*

**Input.** An integer  $n$ .

**Output.** The last digit of the  $n$ -th Fibonacci number.



## Huge Fibonacci Number

*Compute the  $n$ -th Fibonacci number modulo  $m$ .*

**Input.** Integers  $n$  and  $m$ .

**Output.**  $n$ -th Fibonacci number modulo  $m$ .

$$1 + 1 + 2 + 3 + 5 + 8 = 20$$

### Last Digit of the Sum of Fibonacci Numbers

Compute the last digit of  $F_0 + F_1 + \dots + F_n$ .

Input. An integer  $n$ .

Output. The last digit of  $F_0 + F_1 + \dots + F_n$ .

$$2 + 3 + 5 + 8 + 13 = 31$$

### Last Digit of the Partial Sum of Fibonacci Numbers

Compute the last digit of  $F_m + F_{m+1} + \dots + F_n$ .

Input. Integers  $m \leq n$ .

Output. The last digit of  $F_m + F_{m+1} + \dots + F_n$ .

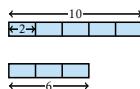


### Last Digit of the Sum of Squares of Fibonacci Numbers

Compute the last digit of  $F_0^2 + F_1^2 + \dots + F_n^2$ .

Input. An integer  $n$ .

Output. The last digit of  $F_0^2 + F_1^2 + \dots + F_n^2$ .

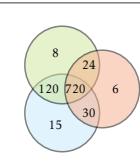


### Greatest Common Divisor

Compute the greatest common divisor of two positive integers.

Input. Two positive integers  $a$  and  $b$ .

Output. The greatest common divisor of  $a$  and  $b$ .



### Least Common Multiple

Compute the least common multiple of two positive integers.

Input. Two positive integers  $a$  and  $b$ .

Output. The least common multiple of  $a$  and  $b$ .



### Money Change

Compute the minimum number of coins needed to change the given value into coins with denominations 1, 5, and 10.

Input. An integer  $money$ .

Output. The minimum number of coins with denominations 1, 5, and 10 that changes  $money$ .



### Maximizing the Value of the Loot

*Find the maximal value of items that fit into the backpack.*

**Input.** The capacity of a backpack  $W$  as well as the weights  $(w_1, \dots, w_n)$  and costs  $(c_1, \dots, c_n)$  of  $n$  different compounds.

**Output.** The maximum total value of fractions of items that fit into the backpack of the given capacity: i.e., the maximum value of  $c_1 \cdot f_1 + \dots + c_n \cdot f_n$  such that  $w_1 \cdot f_1 + \dots + w_n \cdot f_n \leq W$  and  $0 \leq f_i \leq 1$  for all  $i$  ( $f_i$  is the fraction of the  $i$ -th item taken to the backpack).

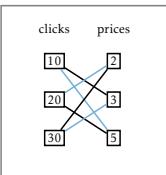


### Car Fueling

*Compute the minimum number of gas tank refills to get from one city to another.*

**Input.** Integers  $d$  and  $m$ , as well as a sequence of integers  $stop_1 < stop_2 < \dots < stop_n$ .

**Output.** The minimum number of refills to get from one city to another if a car can travel at most  $m$  miles on a full tank. The distance between the cities is  $d$  miles and there are gas stations at distances  $stop_1, stop_2, \dots, stop_n$  along the way. We assume that a car starts with a full tank.

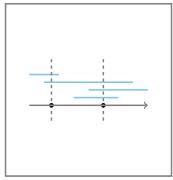


### Maximum Product of Two Sequences

*Find the maximum dot product of two sequences of numbers.*

**Input.** Two sequences of  $n$  positive integers:  $price_1, \dots, price_n$  and  $clicks_1, \dots, clicks_n$ .

**Output.** The maximum value of  $price_1 \cdot c_1 + \dots + price_n \cdot c_n$ , where  $c_1, \dots, c_n$  is a permutation of  $clicks_1, \dots, clicks_n$ .

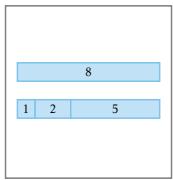


### Covering Segments by Points

*Find the minimum number of points needed to cover all given segments on a line.*

**Input.** A sequence of  $n$  segments  $[l_1, r_1], \dots, [l_n, r_n]$  on a line.

**Output.** A set of points of minimum size such that each segment  $[l_i, r_i]$  contains a point, i.e., there exists a point  $x$  from this set such that  $l_i \leq x \leq r_i$ .



### Distinct Summands

*Represent a positive integer as the sum of the maximum number of pairwise distinct positive integers.*

**Input.** A positive integer  $n$ .

**Output.** The maximum  $k$  such that  $n$  can be represented as the sum  $a_1 + \dots + a_k$  of  $k$  distinct positive integers.

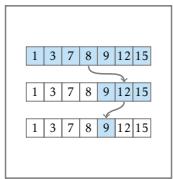


### Largest Concatenate

*Compile the largest number by concatenating the given numbers.*

**Input.** A sequence of positive integers.

**Output.** The largest number that can be obtained by concatenating the given integers in some order.

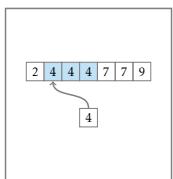


### Sorted Array Multiple Search

*Search multiple keys in a sorted sequence of keys.*

**Input.** A sorted array  $K$  of distinct integers and an array  $Q = \{q_0, \dots, q_{m-1}\}$  of integers.

**Output.** For each  $q_i$ , check whether it occurs in  $K$ .

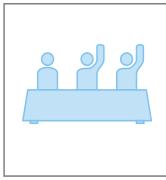


### Binary Search with Duplicates

*Find the index of the first occurrence of a key in a sorted array.*

**Input.** A sorted array of integers (possibly with duplicates) and an integer  $q$ .

**Output.** Index of the first occurrence of  $q$  in the array or “ $-1$ ” if  $q$  does not appear in the array.

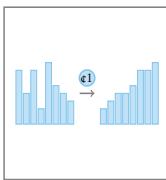


### Majority Element

Check whether a given sequence of numbers contains an element that appears more than half of the times.

Input. A sequence of  $n$  integers.

Output. 1, if there is an element that is repeated more than  $n/2$  times, and 0 otherwise.

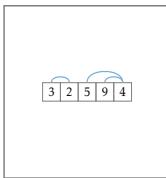


### Speeding-up RANDOMIZEDQUICKSORT

Sort a given sequence of numbers (that may contain duplicates) using a modification of RANDOMIZEDQUICKSORT that works in  $O(n \log n)$  expected time.

Input. An integer array with  $n$  elements that may contain duplicates.

Output. Sorted array (generated using a modification of RANDOMIZEDQUICKSORT) that works in  $O(n \log n)$  expected time.

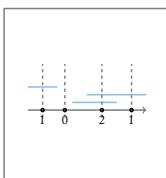


### Number of Inversions

Compute the number of inversions in a sequence of integers.

Input. A sequence of  $n$  integers  $a_1, \dots, a_n$ .

Output. The number of inversions in the sequence, i.e., the number of indices  $i < j$  such that  $a_i > a_j$ .

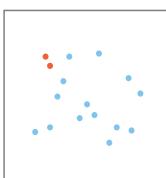


### Points and Segments

Given a set of points and a set of segments on a line, compute, for each point, the number of segments it is contained in.

Input. A list of  $n$  segments and a list of  $m$  points.

Output. The number of segments containing each point.

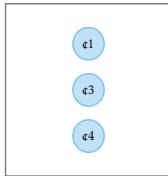


### Closest Points

Find the closest pair of points in a set of points on a plane.

Input. A list of  $n$  points on a plane.

Output. The minimum distance between a pair of these points.

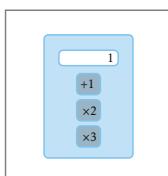


### Money Change Again

Compute the minimum number of coins needed to change the given value into coins with denominations 1, 3, and 4.

Input. An integer *money*.

Output. The minimum number of coins with denominations 1, 3, and 4 that changes *money*.

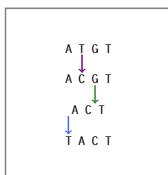


### Primitive Calculator

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

Input. An integer *n*.

Output. The minimum number of operations “+1”, “×2”, and “×3” needed to get *n* from 1.

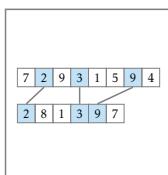


### Edit Distance

Compute the edit distance between two strings.

Input. Two strings.

Output. The minimum number of single-symbol insertions, deletions, and substitutions to transform one string into the other one.

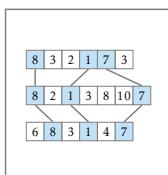


### Longest Common Subsequence of Two Sequences

Compute the maximum length of a common subsequence of two sequences.

Input. Two sequences.

Output. The maximum length of a common subsequence.

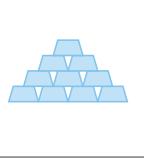


### Longest Common Subsequence of Three Sequences

Compute the maximum length of a common subsequence of three sequences.

Input. Three sequences.

Output. The maximum length of a common subsequence.



### Maximum Amount of Gold

Given a set of gold bars of various weights and a backpack that can hold at most  $W$  pounds, place as much gold as possible into the backpack.

Input. A set of  $n$  gold bars of integer weights  $w_1, \dots, w_n$  and a backpack that can hold at most  $W$  pounds.

Output. A subset of gold bars of maximum total weight not exceeding  $W$ .



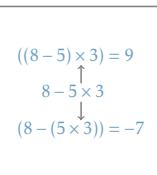
### 3-Partition

Partition a set of integers into three subsets with equal sums.

Input. A sequence of integers  $v_1, v_2, \dots, v_n$ .

Output. Check whether it is possible to partition them into three subsets with equal sums, i.e., check whether there exist three disjoint sets  $S_1, S_2, S_3 \subseteq \{1, 2, \dots, n\}$  such that  $S_1 \cup S_2 \cup S_3 = \{1, 2, \dots, n\}$  and

$$\sum_{i \in S_1} v_i = \sum_{j \in S_2} v_j = \sum_{k \in S_3} v_k.$$



### Maximum Value of an Arithmetic Expression

Parenthesize an arithmetic expression to maximize its value.

Input. An arithmetic expression consisting of digits as well as plus, minus, and multiplication signs.

Output. Add parentheses to the expression in order to maximize its value.

# Interactive Algorithmic Puzzles

You are also welcome to solve the following interactive algorithmic puzzles.



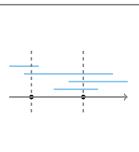
## Eight Queens

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



## 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.



## Covering Segments by Points

Find the minimum number of points that cover all given segments on a line.



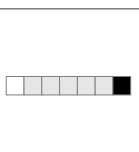
## Activity Selection

Select as many non-overlapping segments as possible.



## Largest Concatenate Problem

Compile the largest number by concatenating the given numbers.



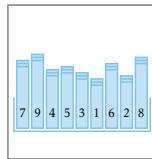
## Black and White Squares

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.



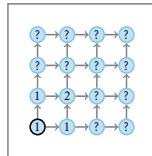
## Twenty One Questions Game

Find an unknown integer  $1 \leq x \leq N$  by asking the minimum number of questions “Is  $x = y?$ ” (for any  $1 \leq y \leq N$ ). Your opponent will reply either “Yes”, or “ $x < y$ ”, or “ $x > y$ .”



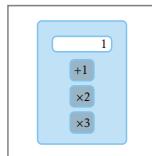
### Book Sorting

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



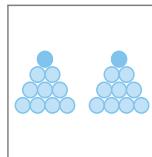
### Number of Paths

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.



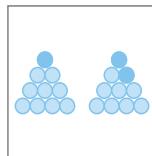
### Antique Calculator

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.



### 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.



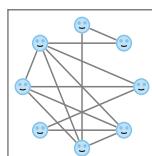
### Three Rocks Game

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.



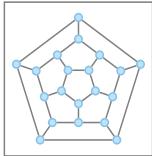
### Map Coloring

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



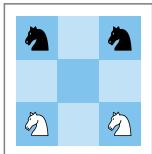
### Clique Finding

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



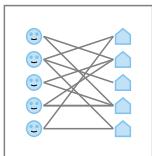
### Icosian Game

Find a cycle visiting each node exactly once.



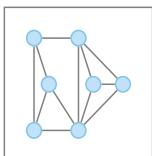
### Guarini Puzzle

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.



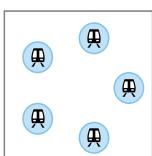
### Room Assignment

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).



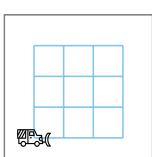
### Tree Construction

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



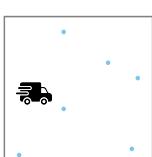
### Subway Lines

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?



### Plow Truck

A plow truck needs to clean every street and get back to its original location. Help it find the shortest route for this.



### Delivery Van

A delivery van needs to visit all the points and get back to the initial location. Help it find the shortest route for this.

# 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** (left) 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 is the head of B.Sc. program “Modern Software Engineering” at St. Petersburg State University and a board member of Computer Science Center in St. 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** (right) 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 (ACM) Fellow (2010), an International Society for Computational Biology (ISCB) Fellow (2012), and a Member of the the Academia Europaea (2016). He was awarded the Senior Scientist Award from ISCB (2017) and the Kanellakis Theory and Practice Award from ACM (2019). 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”, “Analyze Your Genome!”, and “Hacking COVID-19” that are available at Coursera and edX.

## Meet Our Online Co-Instructors



**Daniel Kane** (left) 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** (middle) 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** (right) 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.

We are indebted to Ivan Pavlov for transforming our text into interactive textbook on Stepik.

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 and Darya Borisyak for leading the development of interactive puzzles.

We thank Alexey Kladov, Sergey Lebedev, Alexei Levin, Sergey Shulman, and Alexander Smal for help with the “Best 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.



# Chapter 1: Algorithms and Complexity

This book presents algorithmic programming challenges and puzzles 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 1 000 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.

**Stop and Think.** What is the minimum number of coins of denominations (25, 10, 5, 1) needed to change 77 cents?

The example of 77 cents represents an instance of the Change Problem that assumes that there are  $d$  denominations represented by an array  $c = (c_1, c_2, \dots, c_d)$ . 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  $c = (c_1, c_2, \dots, c_d)$ , in decreasing order of value ( $c_1 > c_2 > \dots > c_d$ ).

**Output:** A list of  $d$  integers  $i_1, i_2, \dots, i_d$  such that  $c_1 \cdot i_1 + c_2 \cdot i_2 + \dots + c_d \cdot i_d = money$ , and  $i_1 + i_2 + \dots + i_d$  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  $money > 0$ :

$coin \leftarrow$  coin with the largest denomination that does not exceed  $money$   
give coin with denomination  $coin$  to customer

$money \leftarrow money - coin$

Here is a faster version of CHANGE:

```

CHANGE(money, c, d):
  r  $\leftarrow$  money
  for k from 1 to d:
    ik  $\leftarrow$   $\lfloor \frac{r}{c_k} \rfloor$ 
    r  $\leftarrow$  r  $- c_k \cdot i_k$ 
  return (i1, i2, ..., id)

```

## 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.

**Stop and Think.** What is the minimum number of coins of denominations (25, 20, 10, 5, 1) needed to change 40 cents?

`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  $c_1, c_2, \dots, c_d$  that adds to *money*, and return the combination with the fewest. We only need to consider combinations with  $i_1 \leq \text{money}/c_1$  and  $i_2 \leq \text{money}/c_2$  (in general,  $i_k$  should not exceed  $\text{money}/c_k$ ), because we would otherwise be returning an amount of money larger than *money*. The pseudocode below uses the symbol  $\sum$  that stands for summation:  $\sum_{i=1}^m a_i = a_1 + a_2 + \dots + a_m$ . The pseudocode also uses the notion of “infinity” (denoted as  $\infty$ ) 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  $\leftarrow \infty$ 
    for each  $(i_1, \dots, i_d)$  from  $(0, \dots, 0)$  to  $(\frac{\text{money}}{c_1}, \dots, \frac{\text{money}}{c_d})$ :
        valueOfCoins  $\leftarrow \sum_{k=1}^d i_k \cdot c_k
        if valueOfCoins = M:
            numberOfCoins =  $\sum_{k=1}^d i_k$ 
            if numberOfCoins < smallestNumberOfCoins:
                smallestNumberOfCoins  $\leftarrow \text{numberOfCoins}$ 
                change  $\leftarrow (i_1, i_2, \dots, i_d)$ 
    return change$ 
```

The second line iterates over every feasible combination  $(i_1, \dots, i_d)$  of the  $d$  indices, and stops when it has reached

$$\left( \frac{\text{money}}{c_1}, \dots, \frac{\text{money}}{c_d} \right).$$

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.** Roughly how many iterations of the for loop does BRUTEFORCECHANGE perform?

- $\text{money}$
- $\text{money}^d$
- $d$

## 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  $n^2$  operations on an input of size  $n$ , and an algorithm  $B$  solves the same problem in  $3n + 2$  operations. Which algorithm,  $A$  or  $B$ , is faster? Although  $A$  may be faster than  $B$  for some small  $n$  (e.g., for  $n$  between 1 and 3),  $B$  will become faster for large  $n$  (e.g., for all  $n > 4$ ). See Figure 1.1. Since  $f(n) = n^2$  is, in some sense, a “faster-growing” function than  $g(n) = n$  with respect to  $n$ , the constants 3 and 2 in  $3n + 2$  do not affect the competition between the two algorithms for large  $n$ . We refer to  $A$  as a *quadratic* algorithm and to  $B$  as a *linear* 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  $(c_1, c_2, \dots, c_d)$ . To

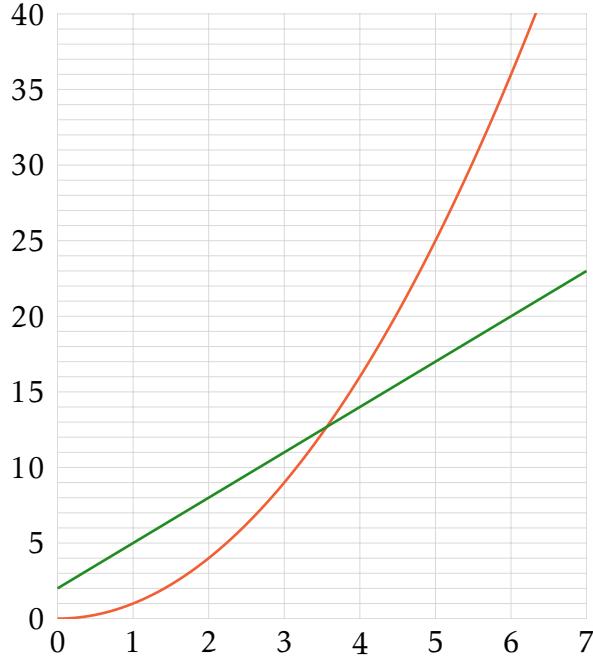


Figure 1.1: Plots of  $n^2$  and  $3n + 2$ .

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

$$\frac{\text{money}}{c_1} \times \frac{\text{money}}{c_2} \times \dots \times \frac{\text{money}}{c_d}$$

iterations, the for loop performs on the order of  $d \times \frac{\text{money}^d}{c_1 c_2 \dots c_d}$  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.,  $c = (1, 2, 3, 4, 5, \dots, 100)$ ), we see that BRUTEFORCECHANGE can take a very long time to execute.

## 1.6 Big-O Notation

To figure out how long a program would take to run on a real computer, we would need to know things like: speed of the computer, the system architecture, the compiler being used, details of the memory hierarchy, etc. Hence, carefully estimating the running time is a rather difficult task. Moreover, in practice, you might not even know some of these details. That is why computer scientists use the *big-O* notation to estimate the running time of an algorithm without knowing anything about all these details!

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

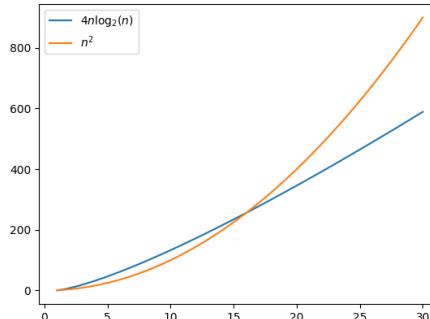
the worst case efficiency of an algorithm is that we are guaranteed that our algorithm will never behave worse than our worst case estimate, so we are never surprised or disappointed. Thus, when we derive a big- $O$  bound, it is a bound on the worst case efficiency.

Consider two algorithms and denote by  $f(n)$  and  $g(n)$  their running times on an input of size  $n$ . We say that  $f$  grows no faster than  $g$ , if there exists a constant  $c$  such that for every positive integer  $n$ ,  $f(n) \leq c \cdot g(n)$  (equivalently,  $\frac{f(n)}{g(n)} \leq c$  for all  $n$ ). In this case, we write  $f = O(g)$  or  $f \leq g$ . The notation  $f = O(g)$  is the standard one, whereas some learners find the notation  $f \leq g$  to be more intuitive.

To give an example, let's visualize functions  $f(n) = 4n \log_2 n$  and  $g(n) = n^2$  by plotting them for  $1 \leq n \leq 30$  using a simple Python code. You can use this code as a template for plotting any other functions.

```
import matplotlib.pyplot as plt
import numpy as np

n = np.linspace(1, 30)
plt.plot(n, 4 * n * np.log2(n), label='$4n\log_2(n)$')
plt.plot(n, n ** 2, label='$n^2$')
plt.legend()
plt.savefig('plot_nlogn_and_n2.png')
```



As the picture reveals,  $4n \log_2 n \geq n^2$  for  $n \leq 16$ , but then the two functions switch. Indeed,  $4 \log_2 n \leq n$  for  $n \geq 16$ . If we set  $c = 16$ , then

$$4n \log_2 n \leq cn^2 \text{ for all } n \geq 1.$$

We conclude that  $4n \log_2 n = O(n^2)$ .

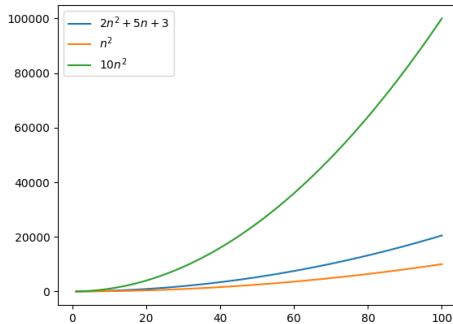
Now, let  $f(n) = 2n^2 + 5n + 3$  and  $g(n) = n^2$ . On the one hand,  $f(n)$  is larger than  $g(n)$  for all positive  $n$ . On the other hand,

$$f(n) = 2n^2 + 5n + 3 \leq 2n^2 + 5n^2 + 3n^2 = 10n^2 = 10g(n)$$

for all positive integers  $n$ . That is,  $f(n)$  is at most ten times larger than  $g(n)$ . We conclude that  $f$  grows no faster than  $g$  (and write  $f = O(g)$  or  $f \leq g$ ).

```
import matplotlib.pyplot as plt
import numpy as np

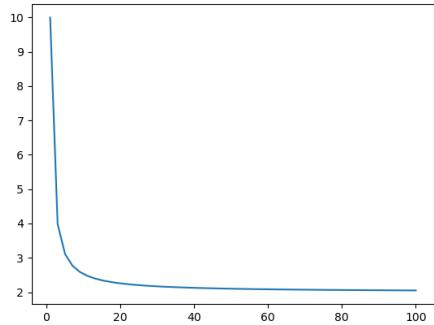
n = np.linspace(1, 100)
plt.plot(n, 2 * n ** 2 + 5 * n + 3, label='$2n^2+5n+3$')
plt.plot(n, n ** 2, label='$n^2$')
plt.plot(n, 10 * n ** 2, label='$10n^2$')
plt.legend(loc='upper left')
plt.savefig('plot_two_squares.png')
```



One can also plot the fraction  $\frac{f(n)}{g(n)}$ .

```
import matplotlib.pyplot as plt
import numpy as np

n = np.linspace(1, 100)
plt.plot(n, (2 * n ** 2 + 5 * n + 3) / (n ** 2))
plt.savefig('plot_two_squares_fraction.png')
```

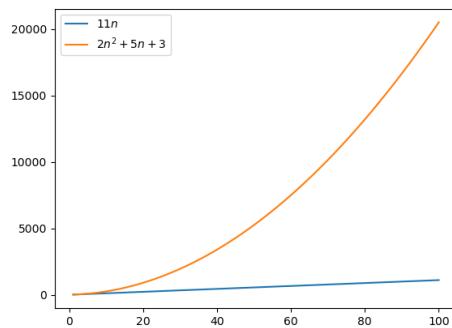


This plot shows that (at least, in the considered range  $1 \leq n \leq 100$ ), the fraction  $\frac{f(n)}{g(n)}$  is at most 10 and, in fact, approaches 2 as  $n$  grows.

Now, let us compare  $f(n) = 11n$  and  $g(n) = 2n^2 + 5n + 3$ . First, let us look at their plots.

```
import matplotlib.pyplot as plt
import numpy as np

n = np.linspace(1, 100)
plt.plot(n, 11 * n, label='$11n$')
plt.plot(n, 2 * n ** 2 + 5 * n + 3, label='$2n^2+5n+3$')
plt.legend()
plt.savefig('plot_square_and_linear.png')
```



This plot reveals that both functions grow (as  $n$  grows) but  $11n$  grows “slower”. This can be formally stated as follows.

For two functions  $f, g$  we say that  $f$  grows slower than  $g$  and write  $f = o(g)$  or  $f < g$ , if the fraction  $\frac{f(n)}{g(n)}$  goes to zero as  $n$  grows.

**Exercise Break.** Plot the fraction  $\frac{11n}{2n^2+5n+3}$  to ensure that it goes to zero as  $n$  grows.

Of course, if  $f < g$  (equivalently,  $f = o(g)$ ), then also  $f \leq g$  (equivalently,  $f = O(g)$ ). In plain English: if  $f$  grows slower than  $g$ , then certainly  $f$  grows no faster than  $g$ .

### 1.6.1 Advantages and Disadvantages

Using the big- $O$  notation to evaluate running times has several advantages:

1. In many cases, computer scientists mainly care about how the running time grows with the input size — the big- $O$  notation clarifies the growth rate.
2. The big- $O$  notation simplifies the formulas for the running time:
  - $O(n^2)$  vs.  $3n^2 + 5n + 2$ .
  - $O(n)$  vs.  $n + \log_2 n + 7$ .
  - $O(n \log n)$  vs  $4n \log_2 n + 5$ . In particular,  $\log_2 n$ ,  $\log_3 n$ , and  $\log_a n$  differ by constant multipliers, so we don't need to specify the base of the logarithm in the big- $O$  notation.
3. With the big- $O$  notation, we no longer need to worry about things like how fast the computer is, or what the memory hierarchy looks like, or what compiler we used. Although these things have a big impact on the final running time, that impact will generally only be a constant multiple.

These advantages come with some disadvantages. Indeed, the big- $O$  notation “loses” some information since it ignores constant multipliers. If you have two algorithms, and one of them is a hundred times faster, they still have the same estimate of the running time in the big- $O$  notation. But, in practice, if you want to make things fast, a factor of 100 is a big deal.

Nevertheless, the big- $O$  notation is very useful and we will use it throughout this book.

## 1.6.2 Five Common Rules

Let us review the common rules of comparing the order of growth of functions arising in algorithm analysis.

1. Multiplicative constants can be omitted:

$$c \cdot f \leq f.$$

Examples:  $5n^2 \leq n^2$ ,  $\frac{n^2}{3} \leq n^2$ ,  $7n \leq n$ .

2. Out of two polynomials, the one with larger degree grows faster:

$$n^a < n^b \text{ for } 0 \leq a < b.$$

Examples:  $n < n^2$ ,  $\sqrt{n} < n^{2/3}$ ,  $n^2 < n^3$ ,  $n^0 < \sqrt{n}$ .

3. Any polynomial grows slower than any exponential:

$$n^a < b^n \text{ for } a \geq 0, b > 1.$$

Examples:  $n^3 < 2^n$ ,  $n^{10} < 1.1^n$ .

4. Any polylogarithm (that is, a function of the form  $(\log n)^a$ ) grows slower than any polynomial:

$$(\log n)^a < n^b \text{ for } a, b > 0.$$

Examples:  $(\log n)^3 < \sqrt{n}$ ,  $n \log n < n^2$ .

5. Smaller terms can be omitted:

$$\text{if } f \leq g, \text{ then } f + g \leq g.$$

Examples:  $n + n^2 \leq n^2$ ,  $n^9 + 2^n \leq 2^n$ .

**Stop and Think.** Mark all correct statements.

- $n^2 \leq (\log_2 n)^{10}$
- $\sqrt{n} \leq n^{5/8}$
- $n^5 + n^4 \leq n^4$

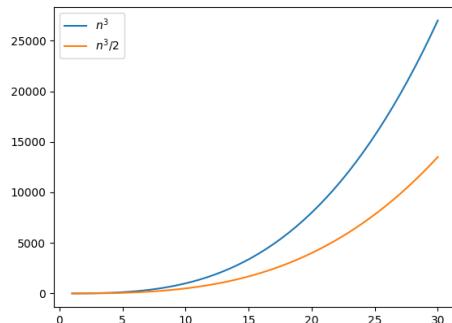
**Stop and Think.** Mark all correct statements.

- $3n + 5 \leq n$
- $n \log_2 n \leq n$
- $\sqrt{n} \leq (\log n)^2$
- $\sqrt{n} \leq n^{3/7}$
- $n^2 \leq 2^n$
- $n + n^2 + n^3 \leq n^2$
- $2^{n+1} \leq 2^n$

### 1.6.3 Visualizing Common Rules

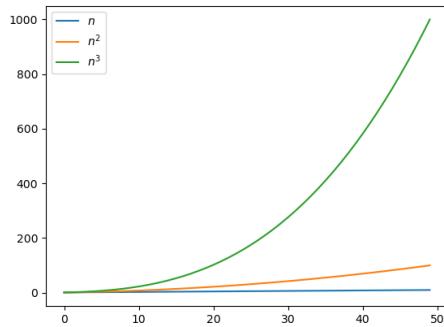
#### 1.6.3.1 Multiplicative Constants Can Be Omitted

The following plot illustrates that that  $n^3$  and  $n^3/2$  have the same growth rate.



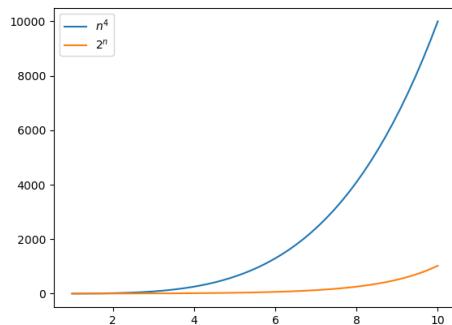
#### 1.6.3.2 Out of Two Polynomials, The One With Larger Degree Grows Faster

The following plot illustrates that  $n^3$  grows faster than  $n^2$  and  $n^2$  grows faster than  $n$ .

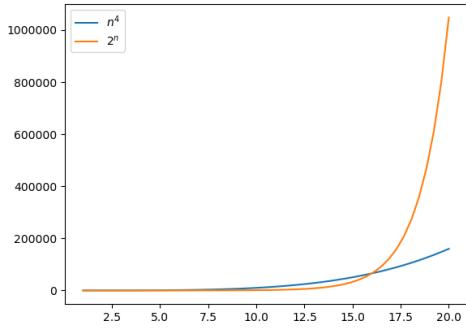


### 1.6.3.3 Any Polynomial Grows Slower Than Any Exponential

The following plot illustrates that  $n^4$  grows faster than  $2^n$  in the range  $1 \leq n \leq 10$ .

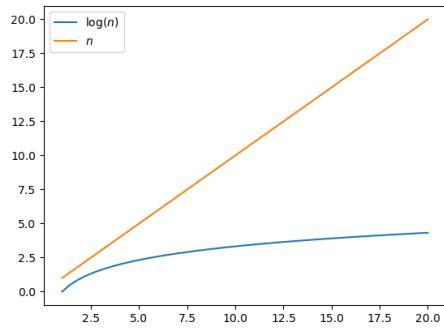


The plot reveals that in this range  $n^4$  is always greater than  $2^n$ . This however does *not* mean that  $n^4$  grows faster than  $2^n$ . Take a look at a larger range  $1 \leq n \leq 20$ .

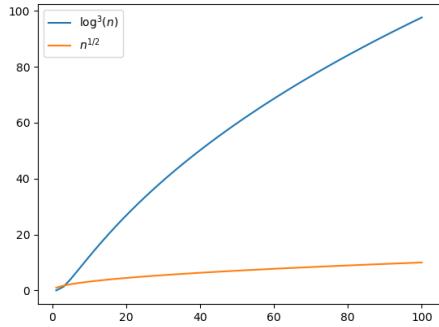


#### 1.6.3.4 Any Polylogarithm Grows Slower Than Any Polynomial

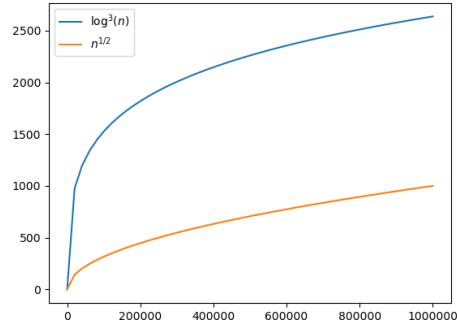
The following plot illustrates that  $\log n$  grows slower than  $n$ .



Now, let us compare  $(\log n)^3$  versus  $\sqrt{n}$ . The following plot illustrates that the polylogarithm  $((\log n)^3)$  grows faster than the polynomial  $n^{1/2}$  in the range  $[1, 100]$ .

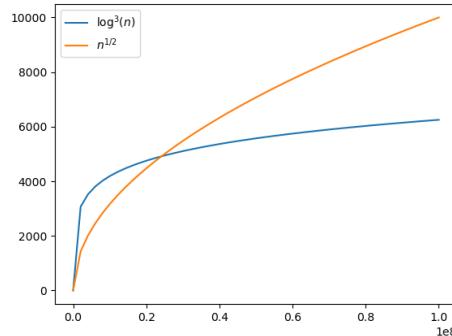


After extending the range from  $[1, 100]$  to  $[1, 1\,000\,000]$ , the polylogarithmic function is still above the polynomial one.

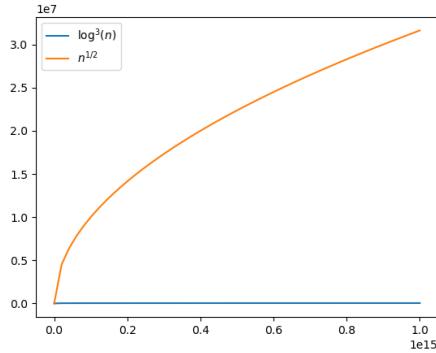


The following plot illustrates that it is in fact dangerous to decide which function grows faster just by looking at how they behave for some values of  $n$ . The rule “any polynomial grows faster than any polylogarithm” means that *eventually* any polynomial will become larger than any polylogarithm. But the rule does not specify for what value of  $n$  this happens for the first time.

To show that  $\sqrt{n}$  grows faster than  $(\log n)^3$ , let us increase the range to  $10^8$ .



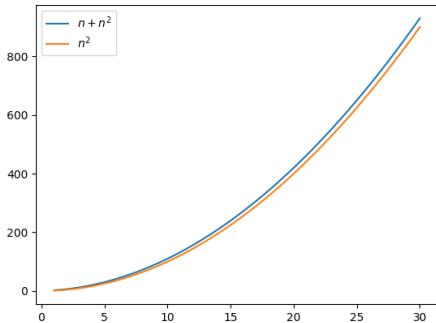
Moreover, let us consider an even larger interval to make sure that these two functions don't switch back.



**Exercise Break.** Find the value of  $n$ , where  $n^{0.1}$  becomes larger than  $(\log n)^5$ .

#### 1.6.3.5 Smaller Terms Can Be Omitted

The plot below shows that the contribution of  $n$  into  $n^2 + n$  is negligible for large  $n$ :  $n + n^2$  and  $n^2$  have the same growth rate.



#### 1.6.4 Frequently Arising Running Times

The table below shows an approximate running time of four algorithms on various input sizes for a hypothetical computer that performs  $10^9$  operations per second. The last row shows the maximum value of  $n$  for which the running time of the corresponding algorithm fits into one second.

	$n$	$n \log n$	$n^2$	$2^n$
$n = 20$	< 1 sec	< 1 sec	< 1 sec	< 1 sec
$n = 50$	< 1 sec	< 1 sec	< 1 sec	13 days
$n = 10^2$	< 1 sec	< 1 sec	< 1 sec	$4 \cdot 10^{13}$ year
$n = 10^6$	< 1 sec	< 1 sec	17 min	
$n = 10^9$	1 sec	30 sec	30 years	
max $n$	$10^9$	$10^{7.5}$	$10^{4.5}$	30

In all programming challenges in this book, your goal is to implement a program that works in at most a few seconds for all allowed input sizes  $n$ . If  $n$  can be as large as  $10^6$ , then you want to design an algorithm with running time  $O(n)$  or  $O(n \log n)$ . If  $n$  is at most  $10^4$ , then you probably want your algorithm to work in time  $O(n^2)$ . And if  $n \leq 20$ , then even an algorithm with the running time  $O(2^n)$  should be OK.

**Stop and Think.** Arrange the following functions in the order of increasing growth rate (the top of the list should contain the function with the slowest growth rate).

1.  $n \log n$
2.  $n^{0.3}$
3.  $2^n$
4.  $\log n$
5.  $n^3$
6.  $\sqrt{n}$
7.  $n^2$

# 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

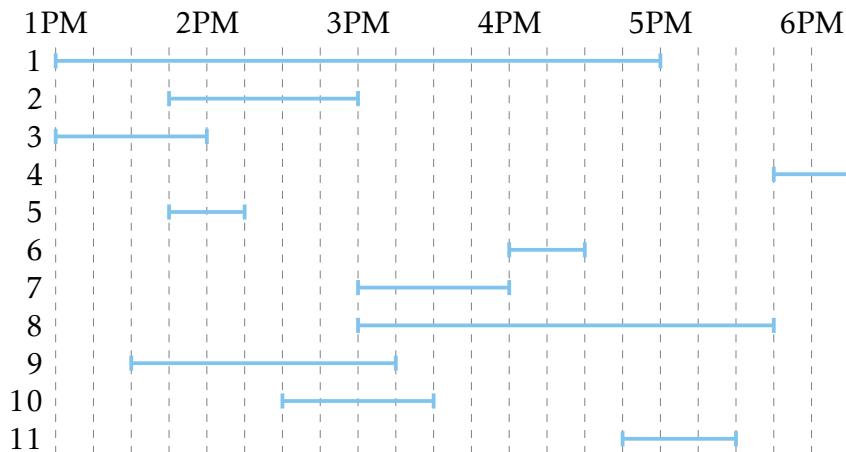
Many algorithms are iterative procedures that choose among a number of alternatives at each iteration. For example, a cashier can view the Change Problem as a series of decisions he or she has to make: which coin (among  $d$  denominations) to return first, which to return second, and so on. Some of 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 strategy would 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. In the Activity Selection Problem ([try it online!](#)), one is given a set of intervals and is asked to select as many of them as possible so that no two of them overlap (two intervals are overlapping if there is a point that belongs to both of them). The name of the problem comes from the following scenario. Imagine a meeting room and the following requests to hold a meeting in this room from eleven groups of people.

1	1:00PM–5:00PM
2	1:45PM–3:00PM
3	1:00PM–2:00PM
4	5:45PM–6:15PM
5	1:45PM–2:15PM
6	4:00PM–4:30PM
7	3:00PM–4:00PM
8	3:00PM–5:45PM
9	1:30PM–3:15PM
10	2:30PM–3:30PM
11	4:45PM–5:30PM

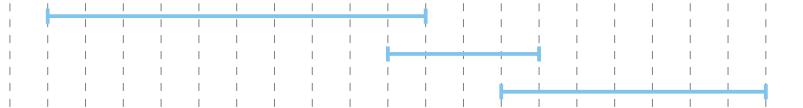
We cannot satisfy all the requests (as some of them are overlapping), but we want to satisfy as many of them as possible. To do this, let us start by visualizing the input data, in a more convenient format.



As we are talking about greedy strategies, let's experiment with various “most profitable” moves here. One common-sense approach is to select the shortest interval, remove all intervals that overlap it, and iterate.

**Stop and Think.** Does this always lead to an optimum solution?

It turns out that this natural greedy strategy can give a suboptimal solution. In the example below, it returns a solution consisting of a single segment (the middle one) whereas there is a solution consisting of two non-overlapping segments.



Another common-sense approach is to select the leftmost interval (that is, the interval with the earliest start time), remove all intervals that overlap it, and iterate.

**Stop and Think.** Does this lead to an optimum solution?

The following example illustrates that this strategy may also fail.



**Stop and Think.** Do you see any other natural greedy strategy for this problem?

It turns out that the following greedy algorithm maximizes the number of non-overlapping segments: repeatedly select an interval with the smallest endpoint (referred to as the *champion* interval), remove all intervals that start before this endpoint, and iterate.

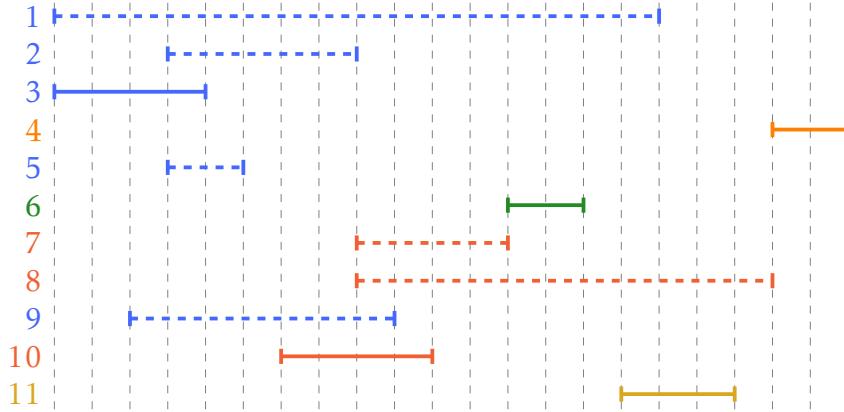
**Exercise Break.** Prove that if a set of non-overlapping intervals does not contain the champion interval then substituting the first interval in this set by the champion interval results in a set of non-overlapping intervals.

This Exercise Break explains why the described greedy strategy is optimal! Indeed, since there exists a solution of the Activity Selection Problem that contains the champion interval, you can safely select the champion interval at the first step and iterate!

For our working example, the algorithm proceeds as follows.

1. Take segment 3 and discard segments 1, 2, 5, and 9.
2. Take segment 10 and discard segments 7 and 8.
3. Take segment 6.
4. Take segment 11.

- Take segment 4.



## 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 ([try it online!](#)) 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.

	0	1	2	3	4	5	6	7	8	9	10
0	<b>W</b>										
1	<b>W</b>	<b>W</b>									
2											
3											
4											
5											
6											
7											
8											
9											
10											

	0	1	2	3	4	5	6	7	8	9	10
0	<b>W</b>	<b>L</b>									
1	<b>W</b>	<b>W</b>									
2	<b>L</b>										
3											
4											
5											
6											
7											
8											
9											
10											

	0	1	2	3	4	5	6	7	8	9	10
0	<b>W</b>	<b>L</b>									
1	<b>W</b>	<b>W</b>	<b>W</b>								
2	<b>L</b>	<b>W</b>									
3											
4											
5											
6											
7											
8											
9											
10											

	0	1	2	3	4	5	6	7	8	9	10
0	<b>W</b>	<b>L</b>	<b>W</b>								
1	<b>W</b>										
2	<b>L</b>	<b>W</b>	<b>L</b>								
3	<b>W</b>										
4	<b>L</b>	<b>W</b>	<b>L</b>								
5	<b>W</b>										
6	<b>L</b>	<b>W</b>	<b>L</b>								
7	<b>W</b>										
8	<b>L</b>	<b>W</b>	<b>L</b>								
9	<b>W</b>										
10	<b>L</b>	<b>W</b>	<b>L</b>								

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

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  $R(2, 2) = L$ , 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( $n, m$ ):
 $R(0, 0) \leftarrow L$ 
for  $i$  from 1 to  $n$ :
    if  $R(i - 1, 0) = W$ :
         $R(i, 0) \leftarrow L$ 
    else:
         $R(i, 0) \leftarrow W$ 
    for  $j$  from 1 to  $m$ :
        if  $R(0, j - 1) = W$ :
             $R(0, j) \leftarrow L$ 
        else:
             $R(0, j) \leftarrow W$ 
    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  $R(i - 1, j) = W$ :
                 $R(i, j) \leftarrow L$ 
            else:
                 $R(i, j) \leftarrow W$ 
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 ([try it online!](#)) and construct the dynamic programming table similar to the table above for this game.

**Stop and Think.** Who wins the Three Rocks game for the  $(7, 5)$  initial position?

- First player.
- Second player.

## 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* ([try it online!](#)) 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  $n$ .

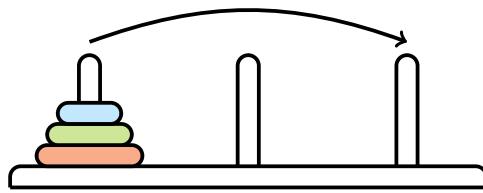
**Output:** A sequence of moves that solve the  $n$ -disk Towers of Hanoi puzzle.

---

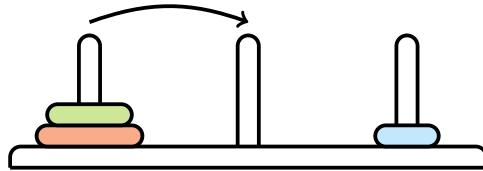
Solving the puzzle with one disk is easy: move the disk to the right peg. The two-disk puzzle is not much harder: move the small disk to the middle peg, then the large disk to the right peg, then the small disk to the right peg to rest on top of the large disk.

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

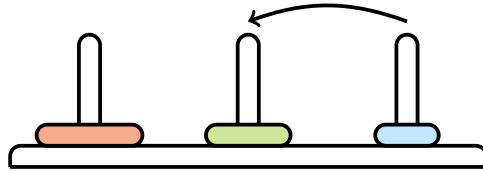
1. Move disk from peg 1 to peg 3



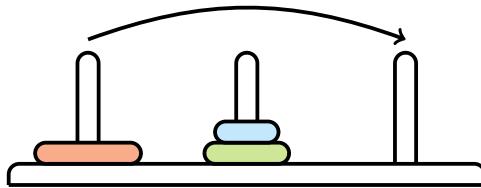
2. Move disk from peg 1 to peg 2



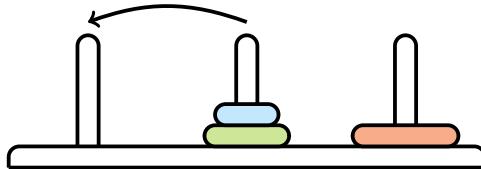
3. Move disk from peg 3 to peg 2



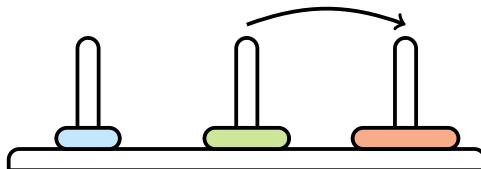
4. Move disk from peg 1 to peg 3



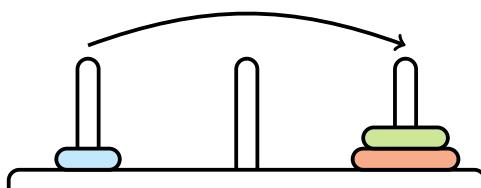
5. Move disk from peg 2 to peg 1



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  $\leftarrow 6 - \text{fromPeg} - \text{toPeg}$ 
HANOITOWERS( $n - 1$ , fromPeg, unusedPeg)
output "Move disk from peg fromPeg to peg toPeg"
HANOITOWERS( $n - 1$ , unusedPeg, toPeg)
```

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  $6 - \text{fromPeg} - \text{toPeg}$ . Table below shows the result of  $6 - \text{fromPeg} - \text{toPeg}$  for all possible values of *fromPeg* and *toPeg*.

<i>fromPeg</i>	<i>toPeg</i>	<i>unusedPeg</i>
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

After computing *unusedPeg* as  $6 - \text{fromPeg} - \text{toPeg}$ , the statements

```
HANOITOWERS( $n - 1$ , fromPeg, unusedPeg)
output "Move disk from peg fromPeg to peg toPeg"
HANOITOWERS( $n - 1$ , unusedPeg, toPeg)
```

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.

**Stop and Think.** How many moves are needed to move 6 disks?

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

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

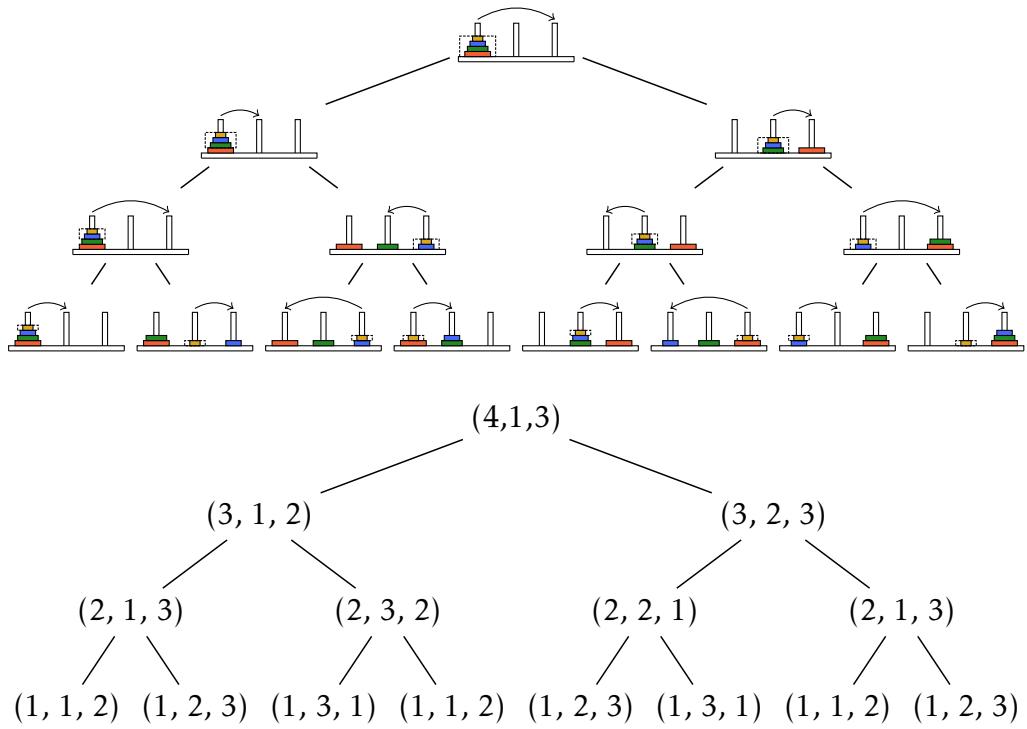


Figure 2.2: The recursion tree for a call to  $\text{HANOITOWERS}(4, 1, 3)$ , which solves the Towers of Hanoi problem of size 4. At each point in the tree,  $(i, j, k)$  stands for  $\text{HANOITOWERS}(i, j, k)$ .

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

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

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

## 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  $a = (a_1, a_2, \dots, a_n)$ .

**Output:** Sorted list of integers, that is, a reordering  $(b_1, b_2, \dots, b_n)$  of integers from  $a$  such that  $b_1 < b_2 < \dots < b_n$ .

---

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.

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)
```

The running time of `SELECTIONSORT` is quadratic, that is,  $O(n^2)$ : there are  $n$  iterations, each requiring time to scan at most  $n$  elements to find the maximum element in the suffix of  $a$ . Note that  $n^2$  is an overestimate of the running time as at the  $i$ -th iteration `SELECTIONSORT` scans a suffix of size  $n - i + 1$ : at the first iteration, it finds the maximum in an array of size  $n$ , at the second iteration, it scans an array of size  $n - 1$ , and so on. Still, the total running time grows as  $n^2$ :

$$n + (n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n(n + 1)}{2}.$$

`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.

$List_1$	2578 2578 2578 2578 2578 2578
$List_2$	346  346  346  346  346  346
$sortedList$	2     3     4     5     6     78

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( $List_1, List_2$ ):
   $SortedList \leftarrow$  empty list
  while both  $List_1$  and  $List_2$  are non-empty:
    if the smallest element in  $List_1$  is smaller than the smallest element in  $List_2$ :
      move the smallest element from  $List_1$  to the end of  $SortedList$ 
    else:
      move the smallest element from  $List_2$  to the end of  $SortedList$ 
  move any remaining elements from either  $List_1$  or  $List_2$  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  $List$ 
   $FirstHalf \leftarrow$  first half of  $List$ 
   $SecondHalf \leftarrow$  second half of  $List$ 
   $SortedFirstHalf \leftarrow$  MERGESORT( $FirstHalf$ )
   $SortedSecondHalf \leftarrow$  MERGESORT( $SecondHalf$ )
   $SortedList \leftarrow$  MERGE( $SortedFirstHalf, SortedSecondHalf$ )
  return  $SortedList$ 

```

**Stop and Think.** What is the running time of MERGESORT?

- $O(n)$
- $O(n \log n)$
- $O(n^2)$

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

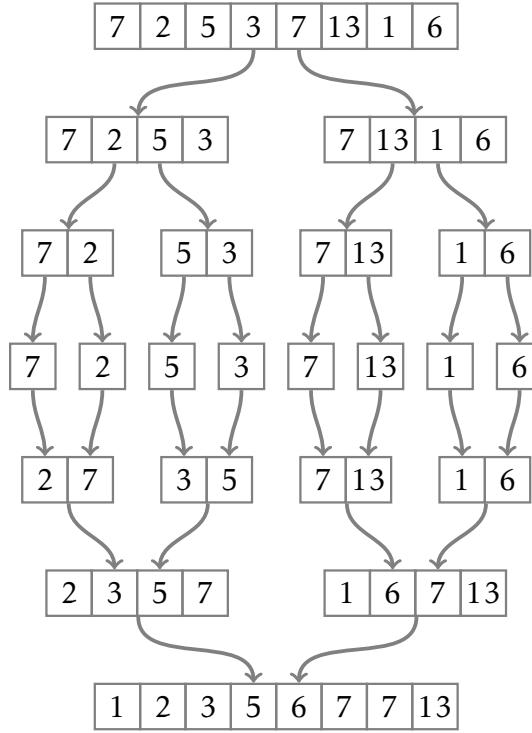


Figure 2.3: The recursion tree for sorting an 8-element array with MERGESORT. The divide (upper) steps consist of  $\log_2 8 = 3$  levels, where the input array is split into smaller and smaller subarrays. The conquer (lower) steps consist of the same number of levels, as the sorted subarrays are merged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

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.

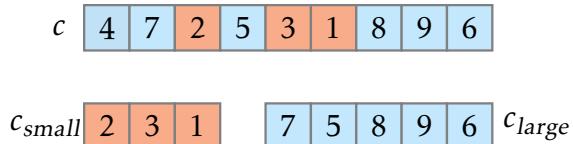
**Exercise Break.** Use the divide-and-conquer strategy to solve our [21 questions](#) interactive puzzle.

## 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.

To give an example of a randomized algorithm, we will first discuss a fast sorting technique called **QUICKSORT**. To simplify its description, assume that all elements of the given array  $c$  are different.

QUICKSORT selects an element  $m$  (e.g., the first) from  $c$  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  $c$ 
 $m \leftarrow$  the first element of  $c$ 
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 sorted array  $c_{sorted}$ 
return  $c_{sorted}$ 

```

The described approach requires allocation of extra memory to store arrays  $c_{small}$  and  $c_{large}$ . A better approach would be to rearrange the elements of the input array *in-place* so that the set  $c_{small}$  appears first, followed by  $m$ , followed by  $c_{large}$  (like shown below), but it is unclear how it can be done.

**Exercise Break.** Nico Lomuto proposed an elegant algorithm for generating the described rearrangement *in-place*. The figure below gives you a hint on how the *Lomuto partition scheme* works. Looking at this Figure, can you reconstruct the logic of Lomuto's approach?

4	7	2	5	3	1	8	9	6
4	7	2	5	3	1	8	9	6
4	2	7	5	3	1	8	9	6
4	2	7	5	3	1	8	9	6
4	2	3	5	7	1	8	9	6
4	2	3	1	7	5	8	9	6
4	2	3	1	7	5	8	9	6
4	2	3	1	7	5	8	9	6
1	2	3	4	7	5	8	9	6

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

$$T(n) = 2T\left(\frac{n}{2}\right) + a \cdot n,$$

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  $(n, n-1, \dots, 2, 1)$ . Worse yet, it requires  $O(n^2)$  time to process  $(1, 2, \dots, n-1, n)$ , 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 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 word “expected” here emphasizes the following effect. Since RANDOMIZEDQUICKSORT is a *randomized* algorithm, two different runs of the algorithm (on the same input) may have different running times: some of them will be quick, some of them will be slow. Thus, 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. It can be shown that, for every array of size  $n$ , the expected running time of RANDOMIZEDQUICKSORT on this array, is  $O(n \log n)$ . A proof can be found in our [slides](#) for Coursera specialization.

The key advantage of randomized algorithms is performance: for many practical problems, randomized algorithms are faster (in the sense of the 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.**  $0 \leq a, b \leq 9$ .

**Sample.**

Input:

9 7

Output:

16

**Time limits (sec.):**

---

C++	Java	Python	C	C#	Go	Haskell	JavaScript	Kotlin	Ruby	Rust	Scala
1	1.5	5	1	1.5	1.5	2	5	1.5	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 sum_of_digits(int first, int second) {  
    return first + second;  
}  
  
int main() {  
    int a = 0;  
    int b = 0;  
    std::cin >> a;  
    std::cin >> b;  
    std::cout << sum_of_digits(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 SumOfDigits {  
    static int sumOfDigits(int first_digit,
```

```

        int second_digit) {
    return first_digit + second_digit;
}

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

```

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

## Python

```

def sum_of_digits(first_digit, second_digit):
    return first_digit + second_digit

if __name__ == '__main__':
    a, b = map(int, input().split())
    print(sum_of_digits(a, b))

```

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

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:** An integer  $n$  and a sequence of  $n$  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		14	8
7	35	42	14		28
4	20	24	8	28	

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

$$\max_{1 \leq i \neq j \leq n} a_i \cdot a_j.$$

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  $a_1, \dots, a_n$  (separated by spaces).

**Output format.** The maximum pairwise product.

**Constraints.**  $2 \leq n \leq 2 \cdot 10^5$ ;  $0 \leq a_1, \dots, a_n \leq 2 \cdot 10^5$ .

#### 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( $A[1 \dots n]$ ):  
product  $\leftarrow 0$   
for  $i$  from 1 to  $n$ :  
    for  $j$  from 1 to  $n$ :  
        if  $i \neq j$ :  
            if  $product < A[i] \cdot A[j]$ :  
                product  $\leftarrow A[i] \cdot A[j]$   
return product
```

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

```
MAXPAIRWISEPRODUCTNAIVE( $A[1 \dots n]$ ):  
product  $\leftarrow 0$   
for  $i$  from 1 to  $n$ :  
    for  $j$  from  $i + 1$  to  $n$ :  
        product  $\leftarrow \max(product, A[i] \cdot A[j])$   
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>  
#include <algorithm>
```

```

int MaxPairwiseProduct(const std::vector<int>& numbers) {
    int max_product = 0;
    int n = numbers.size();

    for (int first = 0; first < n; ++first) {
        for (int second = first + 1; second < n; ++second) {
            max_product = std::max(max_product,
                                   numbers[first] * numbers[second]);
        }
    }

    return max_product;
}

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

    std::cout << MaxPairwiseProduct(numbers) << "\n";
    return 0;
}

```

## Java

```

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

public class MaxPairwiseProduct {
    static int getMaxPairwiseProduct(int[] numbers) {
        int max_product = 0;

```

```

int n = numbers.length;

for (int first = 0; first < n; ++first) {
    for (int second = first + 1; second < n; ++second) {
        max_product = Math.max(max_product,
                               numbers[first] * numbers[second]);
    }
}

return max_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

```

def max_pairwise_product(numbers):
    n = len(numbers)
    max_product = 0
    for first in range(n):
        for second in range(first + 1, n):
            max_product = max(max_product,
                               numbers[first] * numbers[second])

    return max_product

if __name__ == '__main__':
    _ = int(input())
    input_numbers = list(map(int, input().split()))
    print(max_pairwise_product(input_numbers))

```

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( $A[1 \dots n]$ ):
 $index_1 \leftarrow 1$ 
for  $i$  from 2 to  $n$ :
    if  $A[i] > A[index_1]$ :
         $index_1 \leftarrow i$ 
 $index_2 \leftarrow 1$ 
for  $i$  from 2 to  $n$ :
    if  $A[i] \neq A[index_1]$  and  $A[i] > A[index_2]$ :
         $index_2 \leftarrow i$ 
return  $A[index_1] \cdot A[index_2]$ 

```

### 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( $A[1 \dots n]$ ):
 $index_1 \leftarrow 1$ 
for  $i$  from 2 to  $n$ :
    if  $A[i] > A[index_1]$ :
         $index_1 \leftarrow i$ 
if  $index_1 = 1$ :
     $index_2 \leftarrow 2$ 
else:
     $index_2 \leftarrow 1$ 
for  $i$  from 1 to  $n$ :
    if  $A[i] \neq A[index_1]$  and  $A[i] > A[index_2]$ :
         $index_2 \leftarrow i$ 
return  $A[index_1] \cdot A[index_2]$ 

```

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 9000000000. 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

$$2^{31} = 2\,147\,483\,648.$$

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

$$2^{63} = 9\,223\,372\,036\,854\,775\,808.$$

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  $1 \leq i \leq 2 \cdot 10^5$ . 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:

```

STRESTTEST( $N, M$ ):
while true:
     $n \leftarrow$  random integer between 2 and  $N$ 
    allocate array  $A[1\dots n]$ 
    for  $i$  from 1 to  $n$ :
         $A[i] \leftarrow$  random integer between 0 and  $M$ 
        print( $A[1\dots n]$ )
         $result_1 \leftarrow$  MAXPAIRWISEPRODUCTNAIVE( $A$ )
         $result_2 \leftarrow$  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 `STRENGTH(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 100 000 to 9.

**Stop and Think.** Why did we first run `STRENGTH` 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] ≠ A[index1]
```

to

```
i ≠ index1
```

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 1 000 (for larger  $N$ , the naive solution will be pretty slow, because its running time is quadratic). We also change  $M$  to 200 000 and run. We again see the screen filling with words “OK”, wait for a minute, and then decide that (finally!) MAXPAIRWISEPRODUCTFAST is correct. Afterward, 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 problem 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( $A[1 \dots n]$ ):  
    SORT( $A$ )  
    return  $A[n - 1] \cdot A[n]$ 
```

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 ( $2 \leq n \leq 2 \cdot 10^5$ ) 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++	Java	Python	C	C#	Go	Haskell	JavaScript	Kotlin	Ruby	Rust	Scala	
1	1.5	5	1	1.5	1.5	2		5	1.5	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 your laptop performs roughly  $10^8\text{--}10^9$  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  $n = 1\,000$ , and if  $n = 100$ , even an  $O(n^3)$  solution 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, Kotlin, Python2, Python3, Ruby, Rust, 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  $1 \leq n \leq 10^5$ , 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  $0, 0, \dots, 0$  or a sequence  $10^6, 10^6, \dots, 10^6$ . Afterward, 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, 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 your 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.



# Chapter 4: Algorithmic Warm Up

Our goal in this chapter is too show that a clever insight about a problem may make an algorithm for this problem a billion times faster! We will consider a number of programming challenges sharing the following property: a naive algorithm for each such challenge is catastrophically slow. Together, we will design algorithms for these problems that will be as simple as the naive solutions, but much more efficient.

You may think that we are obsessed with the Fibonacci numbers because the first six challenges in this section represent questions about them. However, we start from seemingly similar Fibonacci problems because they gradually become more and more difficult — each new problem requires a development of a new idea. For example, after implementing the “Fibonacci Number” challenge you may be wondering what is so difficult about the “Last Digit of Fibonacci Number”. However, after checking the constraints, you will learn that your implementation of the “Fibonacci Number” will take the eternity to solve the “Last Digit of Fibonacci Number”, forcing you to invent a new idea!

$$F_n = F_{n-1} + F_{n-2}$$

## Fibonacci Number

*Compute the  $n$ -th Fibonacci number.*

Input. An integer  $n$ .

Output.  $n$ -th Fibonacci number.

$$F_{100} = 354\,224\,848\,179$$

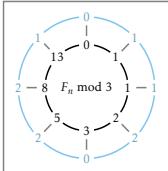
261\,915\,075

## Last Digit of Fibonacci Number

*Compute the last digit of the  $n$ -th Fibonacci number.*

Input. An integer  $n$ .

Output. The last digit of the  $n$ -th Fibonacci number.



## Huge Fibonacci Number

*Compute the  $n$ -th Fibonacci number modulo  $m$ .*

Input. Integers  $n$  and  $m$ .

Output.  $n$ -th Fibonacci number modulo  $m$ .

$$1 + 1 + 2 + 3 + 5 + 8 = 20$$

### Last Digit of the Sum of Fibonacci Numbers

Compute the last digit of  $F_0 + F_1 + \dots + F_n$ .

Input. An integer  $n$ .

Output. The last digit of  $F_0 + F_1 + \dots + F_n$ .

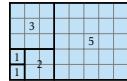
$$2 + 3 + 5 + 8 + 13 = 31$$

### Last Digit of the Partial Sum of Fibonacci Numbers

Compute the last digit of  $F_m + F_{m+1} + \dots + F_n$ .

Input. Integers  $m \leq n$ .

Output. The last digit of  $F_m + F_{m+1} + \dots + F_n$ .

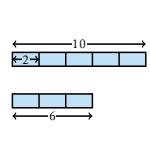


### Last Digit of the Sum of Squares of Fibonacci Numbers

Compute the last digit of  $F_0^2 + F_1^2 + \dots + F_n^2$ .

Input. An integer  $n$ .

Output. The last digit of  $F_0^2 + F_1^2 + \dots + F_n^2$ .

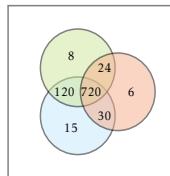


### Greatest Common Divisor

Compute the greatest common divisor of two positive integers.

Input. Two positive integers  $a$  and  $b$ .

Output. The greatest common divisor of  $a$  and  $b$ .



### Least Common Multiple

Compute the least common multiple of two positive integers.

Input. Two positive integers  $a$  and  $b$ .

Output. The least common multiple of  $a$  and  $b$ .

## 4.1 Programming Challenges

### 4.1.1 Fibonacci Number

---

#### Fibonacci Number Problem

Compute the  $n$ -th Fibonacci number.

**Input:** An integer  $n$ .

$$F_n = F_{n-1} + F_{n-2}$$

**Output:**  $n$ -th Fibonacci number.

---

Fibonacci numbers are defined recursively:

$$F_n = \begin{cases} n & \text{if } n \text{ is 0 or 1} \\ F_{n-2} + F_{n-1} & \text{if } n \geq 2 \end{cases}$$

resulting in the following recursive algorithm:

```
FIBONACCI(n):
if n ≤ 1:
    return n
else:
    return FIBONACCI(n - 2) + FIBONACCI(n - 1)
```

**Input format.** An integer  $n$ .

**Output format.**  $F_n$ .

**Constraints.**  $0 \leq n \leq 45$ .

**Sample 1.**

Input:

3

Output:

2

## Sample 2.

Input:

10

Output:

55

**Time and memory limits.** When time/memory limits are not specified, we use the default values specified in Section 3.3.1.

## Solution 1: Recursive Algorithm

Below, we show a straightforward Python implementation of the recursive pseudocode. It contains a debug instruction that prints what is currently computed. We then try to compute  $F_7$  using this code.

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        print(f'Computing F{n} recursively...')
        return fibonacci(n - 2) + fibonacci(n - 1)

print(fibonacci(7))
```

```
Computing F7 recursively...
Computing F5 recursively...
Computing F3 recursively...
Computing F2 recursively...
Computing F4 recursively...
Computing F2 recursively...
Computing F3 recursively...
Computing F2 recursively...
Computing F6 recursively...
Computing F4 recursively...
Computing F2 recursively...
```

```
Computing F3 recursively...
Computing F2 recursively...
Computing F5 recursively...
Computing F3 recursively...
Computing F2 recursively...
Computing F4 recursively...
Computing F2 recursively...
Computing F3 recursively...
Computing F2 recursively...
13
```

As you see, the code produces the right result (13), but many computations are repeated! If you run this code to compute  $F_{150}$ , the Sun may die before your computer returns the result!

**Stop and Think.** How would you compute  $F_7$  by hand?

### Solution 2: Iterative Algorithm

Most probably, you would write on a piece of paper something like this:

$$\begin{aligned}F_0 &= 0 \\F_1 &= 1 \\F_2 &= 0 + 1 = 1 \\F_3 &= 1 + 1 = 2 \\F_4 &= 1 + 2 = 3 \\F_5 &= 2 + 3 = 5 \\F_6 &= 3 + 5 = 8 \\F_7 &= 5 + 8 = 13\end{aligned}$$

And it makes perfect sense to ask a computer to compute  $F_n$  in the same *iterative* manner:

```

FIBONACCI( $n$ ):
if  $n \leq 1$ :
    return  $n$ 
allocate an array  $F[0..n]$ 
 $F[0] \leftarrow 0$ 
 $F[1] \leftarrow 1$ 
for  $i$  from 2 to  $n$ :
     $F[i] \leftarrow F[i - 2] + F[i - 1]$ 
return  $F[n]$ 

```

This algorithm makes about  $n$  arithmetic operations and works well in practice.

**Stop and Think.** Can one avoid storing the entire array?

As you may have already noticed, one does not actually need to store all Fibonacci numbers: for computing the current number, it suffices to remember the two previous numbers.

```

FIBONACCI( $n$ ):
if  $n \leq 1$ :
    return  $n$ 
previous  $\leftarrow 0$ 
current  $\leftarrow 1$ 
repeat ( $n - 1$ ) times:
    oldPrevious  $\leftarrow$  previous
    previous  $\leftarrow$  current
    current  $\leftarrow$  oldPrevious + current
return current

```

### Solution 3: Memoization

The reason why the recursive algorithm is so slow is that it repeats many identical computations: for example  $\text{FIBONACCI}(7)$  calls  $\text{FIBONACCI}(3)$  five times. Would it better to store  $F_3$  the first time it is computed and use this stored value when we need it instead of computing it from scratch? This simple idea is called *memoization*: when something is computed, store this in a data structure to avoid recomputing this again in the future.

Let's wrap the recursive algorithm with memoization to make it practical.

*table*  $\leftarrow$  associative array (*table*[*i*] will be used to store  $F_i$ )

```
FIBONACCI(n):  
if table[n] is not yet computed:  
    if n  $\leq$  1:  
        table[n]  $\leftarrow$  n  
    else:  
        table[n]  $\leftarrow$  FIBONACCI(n - 2) + FIBONACCI(n - 1)  
return table[n]
```

In contrast to the original recursive algorithm, this one will make at most  $n + 1$  “serious” recursive calls: for each  $0 \leq i \leq n$ , the first call to  $\text{FIBONACCI}(i)$  computes  $F_i$  and stores it in  $\text{table}[i]$ ; then, all further calls to  $\text{FIBONACCI}(i)$  will be just table look-ups.

## Code

```
def fibonacci(n):  
    if n  $\leq$  1:  
        return n  
  
    f = [0] * (n + 1)  
    f[0], f[1] = 0, 1  
  
    for i in range(2, n + 1):  
        f[i] = f[i - 2] + f[i - 1]  
  
    return f[n]  
  
if __name__ == '__main__':  
    input_n = int(input())  
    print(fibonacci(input_n))
```

```
def fibonacci(n):
    current, next = 0, 1
    for _ in range(n):
        current, next = next, current + next

    return current

if __name__ == '__main__':
    input_n = int(input())
    print(fibonacci(input_n))
```

```
table = {}

def fibonacci(n):
    if n not in table:
        if n <= 1:
            table[n] = n
        else:
            table[n] = fibonacci(n - 2) +\
                        fibonacci(n - 1)
    return table[n]

if __name__ == '__main__':
    n = int(input())
    print(fibonacci(n))
```

## 4.1.2 Last Digit of Fibonacci Number

---

### Last Digit of Fibonacci Number Problem

Compute the last digit of the  $n$ -th Fibonacci number.

$$F_{100} = 354224848179$$

$$261915075$$

**Input:** An integer  $n$ .

**Output:** The last digit of the  $n$ -th Fibonacci number.

---

**Input format.** An integer  $n$ .

**Output format.** The last digit of  $F_n$ .

**Constraints.**  $0 \leq n \leq 10^6$ .

**Sample 1.**

Input:

3

Output:

2

$F_3 = 2$ .

**Sample 2.**

Input:

139

Output:

1

$F_{139} = 50\ 095\ 301\ 248\ 058\ 391\ 139\ 327\ 916\ 261$ .

**Sample 3.**

Input:

91239

Output:

6

$F_{91239}$  will take more than ten pages to write down, but its last digit is 6.

**Programming tips:** be careful with integer overflow ([LI1](#)).

### Solution: Take Every Intermediate Step Modulo 10

To solve this problem, let's compute  $F_n$  and simply output its last digit:

```
FIBONACCI_LAST_DIGIT(n):
if n ≤ 1:
    return n
allocate an array F[0..n]
F[0] ← 0
F[1] ← 1
for i from 2 to n:
    F[i] ← F[i - 1] + F[i - 2]
return F[n] mod 10
```

Note that Fibonacci numbers grow fast. For example,

$$F_{100} = 354\,224\,848\,179\,261\,915\,075.$$

Therefore, if you use C++ `int32_t` or `int64_t` types for storing  $F$ , you will quickly hit an integer overflow. If you reach out for arbitrary precision numbers, like Java's `BigInteger`, or Python's built-in integers, you'll notice that the loop runs much slower when the iteration number increases.

**Stop and Think.** The last digit of  $F_{102}$  is 6 and the last digit of  $F_{103}$  is 7. What is the last digit of  $F_{104}$ ?

It is not difficult to see that the last digit of  $F_{104}$  is equal to 3 and is determined completely by the last digits of  $F_{102}$  and  $F_{103}$ . This suggests the way to make our algorithm more practical: instead of computing  $F_n$  and taking its last digit, take *every intermediate step modulo 10*.

Here is the main message of this programming challenge: when you need to compute the result of a sequence of arithmetic operations modulo  $m$ , take the result of every single operation modulo  $m$ . This way, you ensure that the numbers you are working with are small (they fit into standard type in your favorite programming language) and that arithmetic operations are performed quickly on them.

```

FIBONACCI LAST DIGIT( $n$ ):
if  $n \leq 1$ :
    return  $n$ 
allocate an array  $F[0..n]$ 
 $F[0] \leftarrow 0$ 
 $F[1] \leftarrow 1$ 
for  $i$  from 2 to  $n$ :
     $F[i] \leftarrow (F[i - 1] + F[i - 2]) \text{ mod } 10$ 
return  $F[n]$ 

```

## Code

```

def fibonacci_last_digit(n):
    if n <= 1:
        return n

    f = [0] * (n + 1)
    f[0], f[1] = 0, 1

    for i in range(2, n + 1):
        f[i] = (f[i - 2] + f[i - 1]) % 10

    return f[n]

if __name__ == '__main__':
    input_n = int(input())
    print(fibonacci_last_digit(input_n))

```

### 4.1.3 Huge Fibonacci Number

---

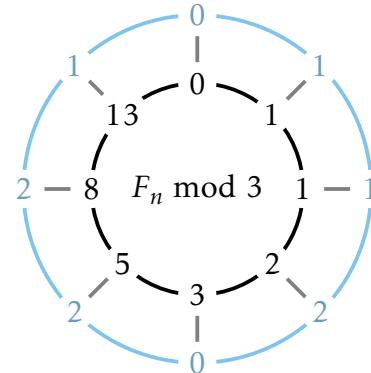
#### Huge Fibonacci Number Problem

Compute the  $n$ -th Fibonacci number modulo  $m$ .

**Input:** Integers  $n$  and  $m$ .

**Output:**  $n$ -th Fibonacci number modulo  $m$ .

---



**Input format.** Integers  $n$  and  $m$ .

**Output format.**  $F_n \bmod m$ .

**Constraints.**  $1 \leq n \leq 10^{14}$ ,  $2 \leq m \leq 10^3$ .

**Sample 1.**

Input:

1 239

Output:

1

$F_1 \bmod 239 = 1 \bmod 239 = 1$ .

**Sample 2.**

Input:

115 1000

Output:

885

$F_{115} \bmod 1000 = 483162952612010163284885 \bmod 1000 = 885$ .

**Sample 3.**

Input:

2816213588 239

Output:

151

$F_{2816213588}$  would require hundreds pages to write it down, but  $F_{2816213588} \bmod 239 = 151$ .

### Solution 1: Pisano Period

In this problem,  $n$  may be so huge that an algorithm going through all Fibonacci numbers  $F_i$  for  $i$  from 0 to  $n$  will be too slow. To get an idea how to solve this problem without going through all  $F_0, F_1, \dots, F_n$ , take a look at the table below:

$i$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$F_i$	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610
$F_i \bmod 2$	0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0
$F_i \bmod 3$	0	1	1	2	0	2	2	1	0	1	1	2	0	2	2	1

**Stop and Think.** Do you see any interesting properties of the last two rows in the table above?

Both these sequences are periodic! For  $m = 2$ , the period is 011 and has length 3, while for  $m = 3$  the period is 01120221 and has length 8.

$i$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$F_i$	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610
$F_i \bmod 2$	0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0
$F_i \bmod 3$	0	1	1	2	0	2	2	1	0	1	1	2	0	2	2	1

Therefore, to compute, say,  $F_{2015} \bmod 3$ , we just need to find the remainder of 2015 when divided by 8. Since  $2015 = 251 \cdot 8 + 7$ , we conclude that  $F_{2015} \bmod 3 = F_7 \bmod 3 = 1$ .

It turns out that for any integer  $m \geq 2$ , the sequence  $F_n \bmod m$  is periodic. The period always starts with 01 and is known as the *Pisano period* (Pisano is another name of Fibonacci).

**Exercise Break.** What is the period of  $F_i \bmod 5$ ?

**Exercise Break.** Prove that  $F_i \bmod m$  is periodic for every  $m$ .

**Exercise Break.** Prove that the period of  $F_i \bmod m$  does not exceed  $m^2$ .

The table below illustrates that the sequence  $F_n \bmod 10$  is periodic — the last digits repeat themselves with the Pisano period of length 60 (see the Fibonacci Number Again Problem). In other words:

$$F_n \bmod 10 = F_{n \bmod 60} \bmod 10.$$

For example,

$$F_{2000} \bmod 10 = F_{2000 \bmod 60} \bmod 10 = F_{20} \bmod 10 = 6765 \bmod 10 = 5.$$

$n$	0	1	2	3	4	5	6	7	...	60	61	62	63	64	65	66	67
$F_n \bmod 10$	0	1	1	2	3	5	8	3	...	0	1	1	2	3	5	8	3

To prove that the sequence of last digits of Fibonacci numbers is periodic, consider pairs of remainders modulo  $m$  of consecutive Fibonacci numbers:

$n$	0	1	2	3	4
$[F_n \bmod m]$	$[0]$	$[1]$	$[1]$	$[2]$	$[3]$
$[F_{n+1} \bmod m]$	$[1]$	$[1]$	$[2]$	$[3]$	$[5]$

Each column in this table can be computed from the previous column  $\begin{bmatrix} a \\ b \end{bmatrix}$  as  $\begin{bmatrix} b \\ (a+b) \bmod m \end{bmatrix}$ . By the same reasoning, the column *before* the column  $\begin{bmatrix} a \\ b \end{bmatrix}$  is  $\begin{bmatrix} (b-a) \bmod m \\ a \end{bmatrix}$ . Hence, any column in the table above can be unambiguously extended both to the left and to the right to fill the entire table.

Since, there are only  $m$  remainders modulo  $m$ , there are only  $m^2$  possible pairs of reminders, i.e., at most  $m^2$  distinct columns. Thus, some columns will eventually repeat in the table and will be repeated forever:

$$\underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix} \dots \begin{bmatrix} a \\ b \end{bmatrix} \dots \begin{bmatrix} a \\ b \end{bmatrix}}_{\text{...}} \underbrace{\begin{bmatrix} a \\ b \end{bmatrix} \dots \begin{bmatrix} a \\ b \end{bmatrix}}_{\text{...}} \underbrace{\begin{bmatrix} a \\ b \end{bmatrix} \dots \begin{bmatrix} a \\ b \end{bmatrix}}_{\text{...}}$$

**Exercise Break.** Prove that the first repeated column in the table for  $F_n \bmod m$  is  $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ .

**Stop and Think.** Why the Pisano period for  $m = 10$  is 60 and not  $10^2 = 100$ , the number of all possible pairs of remainders modulo 10?

This leads us to the following simple pseudocode for computing the Pisano period modulo  $m$  for an arbitrary modulo  $m$ .

```
PISANOPERIOD( $m$ ):
    current  $\leftarrow 0$ 
    next  $\leftarrow 1$ 
    period  $\leftarrow 0$ 
    while True:
        oldNext  $\leftarrow$  next
        next  $\leftarrow$  ( $current + next$ ) mod  $m$ 
        current  $\leftarrow$  oldNext
        period  $\leftarrow$  period + 1
        if  $current = 0$  and  $next = 1$ :
            return period
```

## Code

```
def pisano_period(m):
    current, next = 0, 1
    period = 0

    while True:
        current, next = next, (current + next) % m
        period += 1
        if current == 0 and next == 1:
            return period

def fib_mod(n, m):
    current, next = 0, 1
```

```

for _ in range(n):
    current, next = next, (current + next) % m

return current

if __name__ == '__main__':
    n, m = map(int, input().split())
    print(fib_mod(n % pisano_period(m), m))

```

## Solution 2: Fast Matrix Exponentiation

An alternative way to compute  $F_n \bmod m$  is to notice that the equations

$$\begin{aligned} F_n &= 0 \cdot F_{n-1} + 1 \cdot F_n \\ F_{n+1} &= 1 \cdot F_{n-1} + 1 \cdot F_n \end{aligned}$$

can be represented as multiplying a  $2 \times 2$  matrix  $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$  and a vector  $\begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}$ :

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}.$$

Therefore:

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 \cdot \begin{bmatrix} F_{n-2} \\ F_{n-1} \end{bmatrix} = \dots = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}.$$

Hence,  $F_n$  is simply the top right element of the  $n$ -th power of the matrix  $M = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ .

**Stop and Think.** A naive way to compute  $M^n$  requires  $(n - 1)$  matrix multiplications. Can you do it with only  $O(\log n)$  matrix multiplications?

We illustrate the idea of the *fast matrix exponentiation* using integers rather than matrices. Given an integer  $x$ , a naive way to compute  $x^9$  is

to do 8 multiplications. But here is a faster way to compute  $x^9$  with just 4 multiplications:

$$\begin{aligned}y_1 &= x \cdot x, \\y_2 &= y_1 \cdot y_1, \\y_3 &= y_2 \cdot y_2, \\y_4 &= y_3 \cdot x.\end{aligned}$$

More generally, if  $n$  is even, computing  $x^n$  takes just one more multiplication compared to computing  $y = x^{n/2}$  since  $x^n = x^{n/2} \cdot x^{n/2} = y \cdot y$ . If  $n$  is odd, computing  $x^n$  takes just two more multiplication compared to computing  $y = x^{(n-1)/2}$  since  $x^n = (x^{(n-1)/2} \cdot x^{(n-1)/2}) \cdot x = y \cdot y \cdot x$ .

```
FASTINTEGEREXPONENTIATION( $x, n$ ):  
if  $n = 0$ :  
    return 1  
if  $n$  is even:  
     $z \leftarrow$  FASTINTEGEREXPONENTIATION( $x, n/2$ )  
    return  $z^2$   
else:  
     $z \leftarrow$  FASTINTEGEREXPONENTIATION( $x, (n - 1)/2$ )  
    return  $z^2 \cdot x$ 
```

Since each recursive call `FASTINTEGEREXPONENTIATION` makes two integer multiplications and halves  $n$ , it performs at most  $2 \log n$  multiplications.

Going back to Fibonacci numbers, recall that  $F_n$  is equal to the top right element of  $M^n$ . Since we are interested in the last digit of  $F_n$ , we simply take every intermediate result modulo  $m$ :

```
FASTMATRIXEXPONENTIATION( $D, n, m$ ):  
if  $n = 0$ :  
    return the  $2 \times 2$  identity matrix  
if  $n$  is even:  
     $Z \leftarrow$  FASTMATRIXEXPONENTIATION( $D, n/2, m$ )  
    return MULTIPLY2x2MATRICES( $Z, Z, m$ )  
else:  
     $Z \leftarrow$  FASTMATRIXEXPONENTIATION( $D, n/2, m$ )  
     $Y \leftarrow$  MULTIPLY2x2MATRICES( $Z, Z, m$ )  
    return MULTIPLY2x2MATRICES( $Y, D, m$ )
```

```

MULTIPLY2X2MATRICES( $A, B, m$ ):
 $C_{11} \leftarrow (A_{11} \cdot B_{11} + A_{12} \cdot B_{21}) \bmod m$ 
 $C_{12} \leftarrow (A_{11} \cdot B_{12} + A_{12} \cdot B_{22}) \bmod m$ 
 $C_{21} \leftarrow (A_{21} \cdot B_{11} + A_{22} \cdot B_{21}) \bmod m$ 
 $C_{22} \leftarrow (A_{21} \cdot B_{12} + A_{22} \cdot B_{22}) \bmod m$ 
return  $\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$ 

```

## Code

```

def multiply_2x2_matrices(a, b, m):
    c = [[None] * 2 for _ in range(2)]
    c[0][0] = (a[0][0] * b[0][0] + a[0][1] * b[1][0]) % m
    c[0][1] = (a[0][0] * b[0][1] + a[0][1] * b[1][1]) % m
    c[1][0] = (a[1][0] * b[0][0] + a[1][1] * b[1][0]) % m
    c[1][1] = (a[1][0] * b[0][1] + a[1][1] * b[1][1]) % m
    return c

def matrix_exponent(d, n, m):
    if n == 0:
        return [[1, 0], [0, 1]]
    elif n % 2 == 0:
        z = matrix_exponent(d, n // 2, m)
        return multiply_2x2_matrices(z, z, m)
    else:
        z = matrix_exponent(d, n // 2, m)
        y = multiply_2x2_matrices(z, z, m)
        return multiply_2x2_matrices(y, d, m)

if __name__ == '__main__':
    n, m = map(int, input().split())
    c = [[0, 1], [1, 1]]
    d = matrix_exponent(c, n, m)
    print(d[1][0])

```

#### 4.1.4 Last Digit of the Sum of Fibonacci Numbers

---

##### Last Digit of the Sum of Fibonacci Numbers

###### Problem

Compute the last digit of  $F_0 + F_1 + \dots + F_n$ .

**Input:** An integer  $n$ .

$$1 + 1 + 2 + 3 + 5 + 8 = 20$$

**Output:** The last digit of  $F_0 + F_1 + \dots + F_n$ .

---

Pavel, though this is indeed a math (rather than algorithmic) problem, I would still leave it: it shows again that one can compute something without looping for  $n$  iterations; also, the next problem depends on it

**Input format.** An integer  $n$ .

**Output format.**  $(F_0 + F_1 + \dots + F_n) \bmod 10$ .

**Constraints.**  $0 \leq n \leq 10^{14}$ .

###### Sample 1.

Input:

3

Output:

4

$F_0 + F_1 + F_2 + F_3 = 0 + 1 + 1 + 2 = 4$ .

###### Sample 2.

Input:

100

Output:

5

$F_0 + \dots + F_{100} = 927\,372\,692\,193\,078\,999\,175$ .

**Hint.** Since the brute force approach for this problem is too slow, try to come up with a formula for  $F_0 + F_1 + F_2 + \dots + F_n$ . Play with small values of  $n$  to get an insight and use a solution for the previous problem afterward.

### Solution

The table below shows the first eleven Fibonacci numbers and the first eleven numbers  $S_n = F_0 + F_1 + \dots + F_n$ .

$n$	0	1	2	3	4	5	6	7	8	9	10
$F_n$	0	1	1	2	3	5	8	13	21	34	55
$S_n$	0	1	2	4	7	12	20	33	54	88	143

**Stop and Think.** Do you see any similarities between sequences  $F_0, \dots, F_{10}$  and  $S_0, \dots, S_{10}$ ?

It looks like  $S_n = F_{n+2} - 1$  — let's prove it by induction. This condition certainly holds for the base step ( $n = 0$ ) since  $S_0 = F_2 - 1$ . For the induction step, let's assume that the statement holds for  $0, 1, \dots, n$  and prove it for  $n + 1$ :

$$S_{n+1} = F_0 + F_1 + \dots + F_n + F_{n+1} = S_n + F_{n+1} = F_{n+2} - 1 + F_{n+1} = F_{n+3} - 1.$$

Another way of arriving to the formula  $S_n = F_{n+2} - 1$  is to sum up the equalities

$$\begin{aligned} F_n &= F_{n+2} - F_{n+1} \\ F_{n-1} &= F_{n+1} - F_n \\ F_{n-2} &= F_n - F_{n-1} \\ &\vdots \\ F_2 &= F_4 - F_3 \\ F_1 &= F_3 - F_2 \\ F_0 &= F_2 - F_1 \end{aligned}$$

Since the identically colored terms cancel out, the sum of all terms on the left is  $S_n$  and the sum of all terms on the right is  $F_{n+2} - F_1 = F_{n+2} - 1$ .

Pavel, I've updated the paragraph below

Thus, the problem reduces to finding the last digit of  $F_{n+2} - 1$ . Thanks to the previous problem, we know how to do this quickly: since the Pisano period modulo 10 is equal to 60, we have

$$F_{n+2} \bmod 10 = F_{(n+2) \bmod 60} \bmod 10.$$

## Code

```
def fibonacci_last_digit(n):
    current, next = 0, 1
    for _ in range(n):
        current, next = next, (current + next) % 10

    return current

if __name__ == '__main__':
    n = int(input())
    print((fibonacci_last_digit((n + 2) % 60) + 9) % 10)
```

```
def multiply_2x2_matrices_mod10(a, b):
    c = [[None] * 2 for _ in range(2)]
    c[0][0] = (a[0][0] * b[0][0] + a[0][1] * b[1][0]) % 10
    c[0][1] = (a[0][0] * b[0][1] + a[0][1] * b[1][1]) % 10
    c[1][0] = (a[1][0] * b[0][0] + a[1][1] * b[1][0]) % 10
    c[1][1] = (a[1][0] * b[0][1] + a[1][1] * b[1][1]) % 10
    return c

def matrix_exponent_mod10(d, n):
    if n == 0:
        return [[1, 0], [0, 1]]
    elif n % 2 == 0:
        z = matrix_exponent_mod10(d, n // 2)
        return multiply_2x2_matrices_mod10(z, z)
    else:
        z = matrix_exponent_mod10(d, n // 2)
```

```
y = multiply_2x2_matrices_mod10(z, z)
return multiply_2x2_matrices_mod10(y, d)

if __name__ == '__main__':
    n = int(input())
    m = [[0, 1], [1, 1]]
    p = matrix_exponent_mod10(m, n + 2)
    print((p[1][0] + 9) % 10)
```

### 4.1.5 Last Digit of the Partial Sum of Fibonacci Numbers

---

#### Last Digit of the Partial Sum of Fibonacci Numbers Problem

Compute the last digit of  $F_m + F_{m+1} + \dots + F_n$ .

**Input:** Integers  $m \leq n$ .

$$2 + 3 + 5 + 8 + 13 = 31$$

**Output:** The last digit of  $F_m + F_{m+1} + \dots + F_n$ .

---

**Input format.** Integers  $m$  and  $n$ .

**Output format.**  $(F_m + F_{m+1} + \dots + F_n) \bmod 10$ .

**Constraints.**  $0 \leq m \leq n \leq 10^{14}$ .

**Sample 1.**

Input:

3 7

Output:

1

$$F_3 + F_4 + F_5 + F_6 + F_7 = 2 + 3 + 5 + 8 + 13 = 31.$$

**Sample 2.**

Input:

10 10

Output:

55

$$F_{10} = 55.$$

**Solution**

Pavel, this is a new solution

The sum of a *partial* sum of Fibonacci number is equal to the difference of two partial sums:

$$\begin{aligned} & F_0 + F_1 + F_2 + F_3 + F_4 + F_5 + F_6 + F_7 \\ - & F_0 - F_1 - F_2 \\ = & \quad F_3 + F_4 + F_5 + F_6 + F_7. \end{aligned}$$

More generally,

$$\sum_{i=m}^n F_i = \left( \sum_{i=0}^n F_i \right) - \left( \sum_{i=0}^{m-1} F_i \right).$$

Thanks to the previous problem, we know how to compute the prefix sums quickly.

## Code

```
def fibonacci_last_digit(n):
    current, next = 0, 1
    for _ in range(n):
        current, next = next, (current + next) % 10

    return current

def sum_last_digit(n):
    return (fibonacci_last_digit((n + 2) % 60) + 9) % 10

if __name__ == '__main__':
    m, n = map(int, input().split())
    print((sum_last_digit(n) - sum_last_digit(m - 1)) % 10)
```

#### 4.1.6 Last Digit of the Sum of Squares of Fibonacci Numbers

---

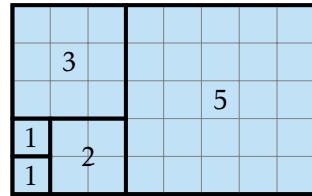
##### Last Digit of the Sum of Squares of Fibonacci Numbers Problem

Compute the last digit of  $F_0^2 + F_1^2 + \dots + F_n^2$ .

**Input:** An integer  $n$ .

**Output:** The last digit of  $F_0^2 + F_1^2 + \dots + F_n^2$ .

---



**Hint.** Since the brute force search algorithm for this problem is too slow ( $n$  may be as large as  $10^{14}$ ), we need to come up with a simple formula for  $F_0^2 + F_1^2 + \dots + F_n^2$ . The figure above represents the sum  $F_1^2 + F_2^2 + F_3^2 + F_4^2 + F_5^2$  as the area of a rectangle with vertical side  $F_5 = 5$  and horizontal side  $F_5 + F_4 = 3 + 5 = F_6$ .

**Input format.** Integer  $n$ .

**Output format.**  $F_0^2 + F_1^2 + \dots + F_n^2 \bmod 10$ .

**Constraints.**  $0 \leq n \leq 10^{14}$ .

**Sample 1.**

Input:

7

Output:

3

$$F_0^2 + F_1^2 + \dots + F_7^2 = 0 + 1 + 1 + 4 + 9 + 25 + 64 + 169 = 273.$$

**Sample 2.**

Input:

73

Output:

1

$$F_0^2 + \cdots + F_{73}^2 = 1\ 052\ 478\ 208\ 141\ 359\ 608\ 061\ 842\ 155\ 201.$$

### Sample 3.

Input:

1234567890

Output:

0

### Solution

Pavel, this is a new solution

The picture above suggests that, for every non-negative integer  $n$ ,

$$F_0^2 + F_1^2 + \cdots + F_n^2 = F_n \cdot F_{n+1}.$$

We prove it by induction. Two base case  $n = 0$  and  $n = 1$  hold:

$$F_0^2 = 0 = F_0 F_1 \text{ and } F_1^2 = 1 = F_1 F_2.$$

For the induction step, assume that  $n \geq 2$ . Then,

$$\begin{aligned} & F_0^2 + F_1^2 + \cdots + F_n^2 + F_{n+1}^2 \\ &= (F_0^2 + F_1^2 + \cdots + F_n^2) + F_{n+1}^2 && \text{(induction step)} \\ &= F_n F_{n+1} + F_{n+1}^2 \\ &= F_{n+1}(F_n + F_{n+1}) \\ &= F_{n+1} F_{n+2}. \end{aligned}$$

Thus, it remains to compute the last digits of  $F_n$  and  $F_{n+1}$ .

### Code

```
def fibonacci_sum_squares(n):
    n = n % 60

    prev, cur = 0, 1
    for _ in range(n):
```

```
    prev, cur = cur, (prev + cur) % 10

    return (prev * cur) % 10

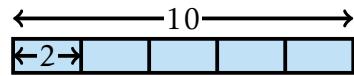
if __name__ == '__main__':
    n = int(input())
    print(fibonacci_sum_squares(n))
```

### 4.1.7 Greatest Common Divisor

---

#### Greatest Common Divisor Problem

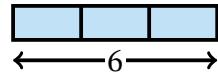
Compute the greatest common divisor of two positive integers.



**Input:** Two positive integers  $a$  and  $b$ .

**Output:** The greatest common divisor of  $a$  and  $b$ .

---



The greatest common divisor  $\text{GCD}(a, b)$  of two positive integers  $a$  and  $b$  is the largest integer  $d$  that divides both  $a$  and  $b$ . The solution of the Greatest Common Divisor Problem was first described (but not discovered!) by the Greek mathematician Euclid twenty three centuries ago. But the name of a mathematician who discovered this algorithm, a century before Euclid described it, remains unknown. Centuries later, Euclid's algorithm was re-discovered by Indian and Chinese astronomers. Now, the efficient algorithm for computing the greatest common divisor is an important ingredient of modern cryptographic algorithms.

Your goal is to implement Euclid's algorithm for computing GCD.

**Input format.** Integers  $a$  and  $b$  (separated by a space).

**Output format.**  $\text{GCD}(a, b)$ .

**Constraints.**  $1 \leq a, b \leq 2 \cdot 10^9$ .

**Sample.**

Input:

28851538 1183019

Output:

17657

$28851538 = 17657 \cdot 1634, 1183019 = 17657 \cdot 67.$

## Naive Algorithms for Computing the Greatest Common Divisor

Here is a simple but terribly slow way to compute the greatest common divisor:

```
GCD( $a, b$ ):  
for  $d$  from  $\min(a, b)$  down to 1:  
    if  $d$  divides both  $a$  and  $b$ :  
        return  $d$ 
```

For example, for  $a = 10^9$  and  $b = 10^9 + 1$ , this algorithm will perform about  $10^9$  divisions.

**Stop and Think.** If you computed  $\text{GCD}(a, b)$ , can you immediately find  $\text{GCD}(a - b, b)$ ?

A simple, but crucial idea that will lead us to a much faster algorithm is suggested by the logo of the programming challenge: it shows that if both  $a$  and  $b$  are divisible by  $d$ , then so is  $a - b$ . It turns out that the converse is also true.

**Exercise Break.** Prove that  $\text{GCD}(a, b) = \text{GCD}(a - b, b)$  for  $a \geq b > 0$ .

This observation allows us to compute the greatest common divisor by subtracting the smaller number from the larger one over and over again. Eventually, one of the numbers will become zero. In this case, we just return the other number (if  $c > 0$ , then  $\text{GCD}(c, 0) = c$ ).

```
GCD( $a, b$ ):  
while  $a > 0$  and  $b > 0$ :  
    if  $a \geq b$ :  
         $a \leftarrow a - b$   
    else:  
         $b \leftarrow b - a$   
return max( $a, b$ )
```

**Exercise Break.** How fast is this algorithm?

This algorithm is still too slow. For example, for  $a = 10^9$  and  $b = 7$ , it keeps subtracting  $b$  from  $a$ , more than a million times (while the original algorithm finds the greatest common divisor for  $a = 10^9$  and  $b = 7$

immediately as it needs to go through  $d = 1, \dots, 7$  only). But not to worry, we will now make our algorithm efficient.

**Stop and Think.** What do we get in the end if we keep subtracting  $b = 7$  from  $a = 10^9$ ?

Right! We get 1, the remainder of  $10^9$  when divided by 7.

**Stop and Think.** If you computed  $\text{GCD}(a, b)$ , can you immediately find  $\text{GCD}(a, b \pmod a)$ ? For example,  $10^9 \pmod 7$  is 1 — what is the relationship between  $\text{GCD}(10^9, 7)$  and  $\text{GCD}(10^9, 1)$ ?

## Euclid's Algorithm

The discussion above leads us to Euclid's algorithm.

```
GCD(a, b):  
while a > 0 and b > 0:  
    if a ≥ b:  
        a ← a mod b  
    else:  
        b ← b mod a  
return max(a, b)
```

This algorithm is fast: for any  $a, b \leq 2 \cdot 10^9$ , it computes their greatest common divisor in a blink of an eye. In particular, for any  $a, b$  in this range, the number of iterations of the while loop is no more than one hundred. This is because at each iteration one of the numbers becomes at least two times smaller.

**Stop and Think.** Prove that for  $a \geq b$ ,  $a \pmod b < a/2$ . Consider two cases:  $b \leq a/2$  and  $b > a/2$ .

Hence, after at most  $\log_2 a + \log_2 b + 2$  iterations, either  $a$  or  $b$  will turn into zero. Since  $a, b \leq 2^{31}$ ,

$$\log_2 a + \log_2 b + 2 < 64.$$

As a final remark we note that the same algorithm can be also implemented recursively in just three lines of code.

```
GCD(a, b):
if a = 0 or b = 0:
    return max(a, b)
return GCD(b, a mod b)
```

## Code

```
def gcd(a, b):
    while a > 0 and b > 0:
        if a >= b:
            a = a % b
        else:
            b = b % a
    return max(a, b)

if __name__ == "__main__":
    a, b = map(int, input().split())
    print(gcd(a, b))
```

```
def gcd(a, b):
    if a == 0 or b == 0:
        return max(a, b)
    return gcd(b, a % b)

if __name__ == "__main__":
    a, b = map(int, input().split())
    print(gcd(a, b))
```

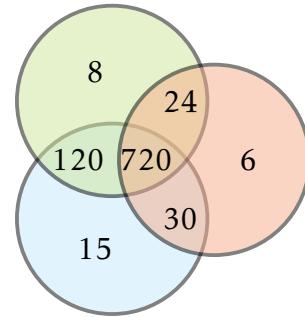
#### 4.1.8 Least Common Multiple

##### Least Common Multiple Problem

Compute the least common multiple of two positive integers.

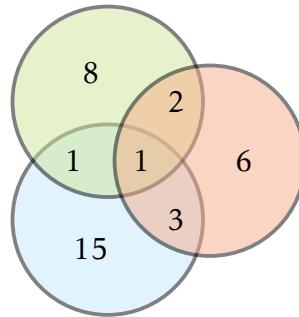
**Input:** Two positive integers  $a$  and  $b$ .

**Output:** The least common multiple of  $a$  and  $b$ .



The least common multiple  $\text{LCM}(a, b)$  of two positive integers  $a$  and  $b$  is the smallest integer  $m$  that is divisible by both  $a$  and  $b$ .

The figure above shows the LCM for each pair of numbers 6, 8, and 15 as well as the LCM for all three of them. The figure below shows the GCD for the same numbers.



**Stop and Think.** How  $\text{LCM}(a, b)$  is related to  $\text{GCD}(a, b)$ ?

**Input format.** Integers  $a$  and  $b$  (separated by a space).

**Output format.**  $\text{LCM}(a, b)$ .

**Constraints.**  $1 \leq a, b \leq 10^7$ .

**Sample.**

Input:

761457 614573

Output:

467970912861

**Programming tips:** use `//` (instead of `/`) for integer division in Python3 ([PG8](#)).

## Code

```
from math import gcd

if __name__ == '__main__':
    a, b = map(int, input().split())
    print(a * b // gcd(a, b))
```

## 4.2 Summary of Algorithmic Ideas

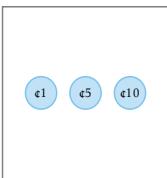
Now when you've implemented several algorithms, let's summarize and review the main ideas that we've discussed in this chapter.

**What solution fits into a second?** Modern computers perform about  $10^8$ – $10^9$  basic operations per second. If your program contains a loop with  $n$  iterations and  $n$  can be as large as  $10^{14}$ , it will run for a couple of days. In turn, this means that for a programming challenge where a naive solution takes as many steps, you need to come up with a different idea.

**Working with large integers.** If you need to compute the last digit of an integer  $m$ , then, in many cases, you can avoid computing  $m$  explicitly: when computing it, take every intermediate step modulo 10. This will ensure that all intermediate values are small enough so that they fit into integer types (in programming languages with integer overflow) and that arithmetic operations with them are fast.

# Chapter 5: Greedy Algorithms

In this chapter, you will learn about seemingly naive yet powerful greedy algorithms. After learning the key idea behind the greedy algorithms, some students feel that they represent the algorithmic Swiss army knife that can be applied to solve nearly all programming challenges in this book. Be warned: since this intuitive idea rarely works in practice, you have to prove that your greedy algorithm produces an optimal solution!



## Money Change

*Compute the minimum number of coins needed to change the given value into coins with denominations 1, 5, and 10.*

Input. An integer *money*.

Output. The minimum number of coins with denominations 1, 5, and 10 that changes *money*.



## Maximizing the Value of the Loot

*Find the maximal value of items that fit into the backpack.*

Input. The capacity of a backpack *W* as well as the weights  $(w_1, \dots, w_n)$  and costs  $(c_1, \dots, c_n)$  of *n* different compounds.

Output. The maximum total value of fractions of items that fit into the backpack of the given capacity: i.e., the maximum value of  $c_1 \cdot f_1 + \dots + c_n \cdot f_n$  such that  $w_1 \cdot f_1 + \dots + w_n \cdot f_n \leq W$  and  $0 \leq f_i \leq 1$  for all *i* ( $f_i$  is the fraction of the *i*-th item taken to the backpack).

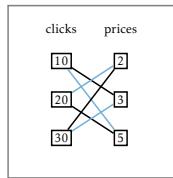


### Car Fueling

Compute the minimum number of gas tank refills to get from one city to another.

Input. Integers  $d$  and  $m$ , as well as a sequence of integers  $stop_1 < stop_2 < \dots < stop_n$ .

Output. The minimum number of refills to get from one city to another if a car can travel at most  $m$  miles on a full tank. The distance between the cities is  $d$  miles and there are gas stations at distances  $stop_1, stop_2, \dots, stop_n$  along the way. We assume that a car starts with a full tank.

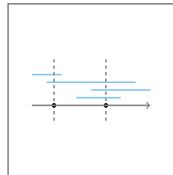


### Maximum Product of Two Sequences

Find the maximum dot product of two sequences of numbers.

Input. Two sequences of  $n$  positive integers:  $price_1, \dots, price_n$  and  $clicks_1, \dots, clicks_n$ .

Output. The maximum value of  $price_1 \cdot c_1 + \dots + price_n \cdot c_n$ , where  $c_1, \dots, c_n$  is a permutation of  $clicks_1, \dots, clicks_n$ .

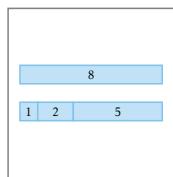


### Covering Segments by Points

Find the minimum number of points needed to cover all given segments on a line.

Input. A sequence of  $n$  segments  $[l_1, r_1], \dots, [l_n, r_n]$  on a line.

Output. A set of points of minimum size such that each segment  $[l_i, r_i]$  contains a point, i.e., there exists a point  $x$  from this set such that  $l_i \leq x \leq r_i$ .



### Distinct Summands

Represent a positive integer as the sum of the maximum number of pairwise distinct positive integers.

Input. A positive integer  $n$ .

Output. The maximum  $k$  such that  $n$  can be represented as the sum  $a_1 + \dots + a_k$  of  $k$  distinct positive integers.



## Largest Concatenate

*Compile the largest number by concatenating the given numbers.*

**Input.** A sequence of positive integers.

**Output.** The largest number that can be obtained by concatenating the given integers in some order.

## 5.1 The Main Idea

### 5.1.1 Examples

A greedy algorithm builds a solution piece by piece and at each step, chooses the most profitable piece. This is best illustrated with examples.

Our first example is the Largest Concatenate Problem: given a sequence of single-digit numbers, find the largest number that can be obtained by concatenating these numbers. For example, for the input sequence  $(2, 3, 9, 1, 2)$ , the output is the number 93221. It is easy to come up with an algorithm for this problem. Clearly, the largest single-digit number should be selected as the first digit of the concatenate. Afterward, we face essentially the same problem: concatenate the remaining numbers to get as large number as possible.

```
LARGESTCONCATENATE(Numbers):  
    result  $\leftarrow$  empty string  
    while Numbers is not empty:  
        maxNumber  $\leftarrow$  largest among Numbers  
        append maxNumber to result  
        remove maxNumber from Numbers  
    return result
```

Our second example is the Money Change Problem: given a non-negative integer *money*, find the minimum number of coins with denominations 1, 5, and 10 that changes *money*. For example, the minimum number of coins needed to change  $money = 28$  is 6:  $28 = 10 + 10 + 5 + 1 + 1 + 1$ . This representation of 28 already suggests an algorithm. We take a coin *c* with the largest denomination that does not exceed *money*. Afterward, we face essentially the same problem: change  $(money - c)$  with the minimum number of coins.

```
CHANGE(money, Denominations):  
    numCoins  $\leftarrow 0  
    while money > 0:  
        maxCoin  $\leftarrow$  largest among Denominations that does not exceed money  
        money  $\leftarrow$  money - maxCoin  
        numCoins  $\leftarrow$  numCoins + 1  
    return numCoins$ 
```

**Exercise Break.** What does `LARGESTCONCATENATE([2, 21])` return?

**Exercise Break.** What does `CHANGE(8, [1, 4, 6])` return?

If you use the same greedy strategy, then `LARGESTCONCATENATE([2, 21])` returns 212 and `CHANGE(8, [1, 4, 6])` returns 3 because  $8 = 6 + 1 + 1$ . But this strategy fails because the correct solutions are 221 (concatenating 2 with 21) and 2 because  $8 = 4 + 4$ !

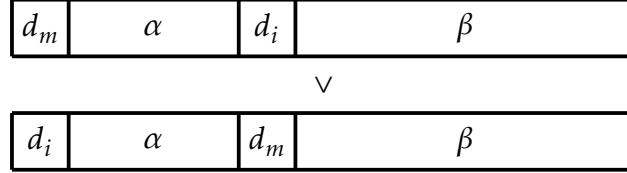
Thus, in *rare* cases when a greedy strategy works, one should be able to prove its correctness: A priori, there should be no reason why a sequence of *locally* optimal moves leads to a *global* optimum!

### 5.1.2 Proving Correctness of Greedy Algorithms

At each step, a greedy algorithm restricts the search space by selecting a most “profitable” piece of a solution. For example, instead of considering all possible concatenations, the `LARGESTCONCATENATE` algorithm only considers concatenations starting from the largest digit. Instead of all possible ways of changing money, the `CHANGE` algorithm considers only the ones that include a coin with the largest denomination (that does not exceed `money`). What one needs to prove is that this restricted search space still contains at least one optimal solution. This is usually done as follows.

Consider an arbitrary optimal solution. If it belongs to the restricted search space, then we are done. If it does not belong to the restricted search space, tweak it so that it is still optimum and belongs to the restricted search space.

Here, we will prove the correctness of `LARGESTCONCATENATE` for single digit numbers (the correctness of `CHANGE` for denominations 1, 5, and 10 will be given in Section 5.2.1). Let  $N$  be the largest number that can be obtained by concatenating digits  $d_1, \dots, d_n$  in some order and let  $d_m$  be the largest digit. Then,  $N$  starts with  $d_m$ . Indeed, assume that  $N$  starts with some other digit  $d_i < d_m$ . Then  $N$  has the form  $d_i\alpha d_m\beta$  where  $\alpha, \beta$  are (possibly empty) sequences of digits. But if we swap  $d_i$  and  $d_m$ , we get a larger number!



**Stop and Think.** What part of this proof breaks for multi-digit numbers?

### 5.1.3 Implementation

A greedy solution chooses the most profitable move and then continues to solve the remaining problem that usually has the same type as the initial one. There are two natural implementations of this strategy: either iterative with a while loop or recursive. Iterative solutions for the Largest Concatenate and Money Change problems are given above. Below are their recursive variants.

```
LARGESTCONCATENATE(Numbers):
if Numbers is empty:
    return empty string
maxNumber ← largest among Numbers
remove maxNumber from Numbers
return concatenate of maxNumber and LARGESTCONCATENATE(Numbers)
```

```
CHANGE(money, Denominations):
if money = 0:
    return 0
maxCoin ← largest among Denominations that does not exceed money
return 1 + CHANGE(money - maxCoin, Denominations)
```

For all the following programming challenges, we provide both iterative and recursive Python solutions.

## 5.2 Programming Challenges

### 5.2.1 Money Change

---

#### Money Change Problem

Compute the minimum number of coins needed to change the given value into coins with denominations 1, 5, and 10.

**Input:** An integer *money*.

**Output:** The minimum number of coins with denominations 1, 5, and 10 that changes *money*.



---

In this problem, you will implement a simple greedy algorithm used by cashiers all over the world. We assume that a cashier has unlimited number of coins of each denomination.

**Input format.** Integer *money*.

**Output format.** The minimum number of coins with denominations 1, 5, 10 that changes *money*.

**Constraints.**  $1 \leq \text{money} \leq 10^3$ .

#### Sample 1.

Input:

2

Output:

2

$2 = 1 + 1$ .

#### Sample 2.

Input:

28

Output:

6

$28 = 10 + 10 + 5 + 1 + 1 + 1$ .

## Solution: Use the Largest Denomination First

Here is the idea: while  $money > 0$ , keep taking a coin with the largest denomination that does not exceed  $money$ , subtracting its value from  $money$ , and adding 1 to the count of the number of coins:

```
CHANGE( $money$ ):  
     $numCoins \leftarrow 0$   
    while  $money > 0$ :  
        if  $money \geq 10$ :  
             $money \leftarrow money - 10$   
        else if  $money \geq 5$ :  
             $money \leftarrow money - 5$   
        else:  
             $money \leftarrow money - 1$   
         $numCoins \leftarrow numCoins + 1$   
    return  $numCoins$ 
```

There is also a one-liner for solving this problem:

```
return  $\lfloor money/10 \rfloor + \lfloor (money \bmod 10)/5 \rfloor + (money \bmod 5)$ 
```

Designing greedy algorithms is easy, but proving that they work is often non-trivial! You are probably wondering why we should waste time proving the correctness of the obvious CHANGE algorithm. Just wait until we setup an algorithmic trap to convince you that the proof below is not a waste of time! To prove that this greedy algorithms is correct, we show that taking a coin with the largest denomination is consistent with some optimal solution. I.e., we need to prove that for any positive integer  $money$  there exists an optimal way of changing  $money$  that uses at least one coin with the denomination  $D$ , where  $D$  is the largest number among 1, 5, 10 that does not exceed  $money$ . We prove this by considering a few cases. In each of the cases, we take some solution (i.e., a particular change for  $money$ ) and transform it so that the number of coins does not increase and it contains at least one coin with denomination  $D$ . In particular, if we start from an *optimal* way to change  $money$ , what we get is also an *optimal* way of changing  $money$  that contains a coin  $D$ .

1.  $1 \leq money < 5$ . In this case  $D = 1$  and the only way to change  $money$  is to use  $money$  coins of denomination 1.

2.  $5 \leq \text{money} < 10$ . In this case  $D = 5$ . Clearly, any change of  $\text{money}$  uses only coins with denominations 1 and 5. If it does not use a coin with denomination 5, then it uses at least five coins of denomination 1 (since  $\text{money} \geq 5$ ). By replacing them with one coin of denomination 5 we improve this solution.
3.  $10 \leq \text{money}$ . In this case  $D = 10$ . Consider a way of changing  $\text{money}$  and assume that it does not use a coin 10. A simple, but crucial observation is that some subset of the used coins sums up to 10. This can be shown by considering the number of coins of denomination 5 in this solution: if there are no 5's, then there are at least ten 1's and we replace them with a single 10; if there is exactly one 5, then there are at least five 1's and we replace them with a single 10 again; if there are at least two 5's, they can be again replaced.

Although this proof is long and rather boring, you need a proof each time you come up with a greedy algorithm! The next Exercise Break hints a more compact way of proving the correctness of the algorithm above.

**Exercise Break.** Show that  $\text{money} \bmod 5$  coins of denomination 1 are needed in any solution and that the rest should be changed with coins of denomination 10 and at most one coin of denomination 5.

**Running time.** The running time of the `CHANGE` algorithm is  $O(\text{money})$ , while its single line version requires only a few arithmetic operations.

## Code

```
def change(money):
    num_coins = 0

    while money > 0:
        if money >= 10:
            money -= 10
        elif money >= 5:
            money -= 5
        else:
```

```
    money -= 1
    num_coins += 1

    return num_coins

if __name__ == "__main__":
    m = int(input())
    print(change(m))
```

```
def change(money):
    return money // 10 + (money % 10) // 5 + (money % 5)

if __name__ == "__main__":
    m = int(input())
    print(change(m))
```

```
def change(money):
    if not money:
        return 0
    max_coin = max([coin for coin in (1, 5, 10)
                   if coin <= money])
    return 1 + change(money - max_coin)

if __name__ == "__main__":
    m = int(input())
    print(change(m))
```

## 5.2.2 Maximum Value of the Loot

### Maximizing the Value of the Loot Problem

*Find the maximal value of items that fit into the backpack.*

**Input:** The capacity of a backpack  $W$  as well as the weights  $(w_1, \dots, w_n)$  and costs  $(c_1, \dots, c_n)$  of  $n$  different compounds.

**Output:** The maximum total value of fractions of items that fit into the backpack of the given capacity: i.e., the maximum value of  $c_1 \cdot f_1 + \dots + c_n \cdot f_n$  such that  $w_1 \cdot f_1 + \dots + w_n \cdot f_n \leq W$  and  $0 \leq f_i \leq 1$  for all  $i$  ( $f_i$  is the fraction of the  $i$ -th item taken to the backpack).



A thief breaks into a spice shop and finds four pounds of saffron, three pounds of vanilla, and five pounds of cinnamon. His backpack fits at most nine pounds, therefore he cannot take everything. Assuming that the prices of saffron, vanilla, and cinnamon are \$5 000, \$200, and \$10, respectively, what is the most valuable loot in this case? If the thief takes  $u_1$  pounds of saffron,  $u_2$  pounds of vanilla, and  $u_3$  pounds of cinnamon, the total value of the loot is

$$5000 \cdot \frac{u_1}{4} + 200 \cdot \frac{u_2}{3} + 10 \cdot \frac{u_3}{5}.$$

The thief would like to maximize the value of this expression subject to the following constraints:  $u_1 \leq 4$ ,  $u_2 \leq 3$ ,  $u_3 \leq 5$ ,  $u_1 + u_2 + u_3 \leq 9$ .

**Input format.** The first line of the input contains the number  $n$  of compounds and the capacity  $W$  of a backpack. The next  $n$  lines define the costs and weights of the compounds. The  $i$ -th line contains the cost  $c_i$  and the weight  $w_i$  of the  $i$ -th compound.

**Output format.** Output the maximum value of compounds that fit into the backpack.

**Constraints.**  $1 \leq n \leq 10^3$ ,  $0 \leq W \leq 2 \cdot 10^6$ ;  $0 \leq c_i \leq 2 \cdot 10^6$ ,  $0 < w_i \leq 2 \cdot 10^6$  for all  $1 \leq i \leq n$ . All the numbers are integers.

**Bells and whistles.** Although the Input to this problem consists of integers, the Output may be non-integer. Therefore, the absolute value of the difference between the answer of your program and the optimal value should be at most  $10^{-3}$ . To ensure this, output your answer with at least four digits after the decimal point (otherwise your answer, while being computed correctly, can turn out to be wrong because of the rounding issues).

### Sample 1.

Input:

```
3 50
60 20
100 50
120 30
```

Output:

```
180.0000
```

To achieve the value 180, the thief takes the entire first compound and the entire third compound.

### Sample 2.

Input:

```
1 10
500 30
```

Output:

```
166.6667
```

The thief should take ten pounds of the only available compound.

**Programming tips:** avoid floating point numbers whenever possible ([LI3](#)).

### Solution: Take as Much of the Most Expensive Compound as You Can

Let's define the *price* of a compound  $i$  as  $c_i/w_i$ . A natural strategy for the thief is to keep taking as much of the priciest (most expensive) compound as possible. To prove that this strategy leads to an optimal solution, let's consider the most expensive compound  $m$ . What is the maximum amount  $a$

of the  $m$ -th compound that the thief can take into his backpack? First, it should fit into the backpack:  $a \leq W$ . Second, it should not exceed the available amount of the  $m$ -th compound:  $a \leq w_m$ . Hence,  $a = \min\{w_m, W\}$ . We claim that there is an optimum solution containing  $a$  pounds of the  $m$ -th compound. To prove it, consider an optimum solution  $u_1, \dots, u_n$  that maximizes the amount  $u_m$  of the most expensive  $m$ -th compound among all optimum solutions ( $u_i$  stands for the amount of the  $i$ -th compound). If  $u_m = a$ , then there is nothing to prove. Otherwise,  $u_m < a$ . Hence,  $u_m < w_m$  and  $u_m < W$ . Consider two cases.

1. The backpack is not fully packed with the current solution:  $u_1 + u_2 + \dots + u_n < W$ . Since  $u_m < w_m$ , one can take a little bit more of the  $m$ -th compound: this way, we get a new solution that is better than the current one, contradicting the optimality of the current one.
2. The backpack is fully packed:  $u_1 + \dots + u_m = W$ . Since  $u_m < W$ , there must be some  $i \neq m$  such that  $u_i > 0$ . Then, instead of some small amount of the  $i$ -th compound, one can take the same small amount of the  $m$ -th compound. This way, we preserve the total weight, but increase the total value and the amount of the most expensive  $m$ -th compound in the backpack. This contradicts the fact that the initial solution had the maximum amount of the  $m$ -th compound.

Having proved that we can take as much of the most expensive compound as we can, we can now design a greedy algorithm: take the maximum possible amount of the most expensive compound and iterate. We stop when either there are no more compounds left or when the backpack is fully packed. In the pseudocode below, *Weight* and *Cost* are arrays that contain all weights and costs.

```

MAXIMUMLOOT( $W, Weight, Cost$ )
if  $W = 0$  or Weight is empty:
    return 0
 $m \leftarrow$  the index of the most expensive item
 $amount \leftarrow \min(Weight[m], W)$ 
 $value \leftarrow Cost[m] \cdot \frac{amount}{Weight[m]}$ 
remove the  $m$ -th element from Weight and Cost
return value + MAXIMUMLOOT( $W, Weight, Cost$ )
```

**Running time.** The running time of this algorithm is  $O(n^2)$ : at every iteration, we scan the list of compounds to find the most expensive one and there are at most  $n$  iterations as every iteration reduces the number of compounds to consider.

## Code

```
import sys

def maximum_loot(capacity, costs, values):
    if not capacity or not costs:
        return 0

    m = 0
    for i in range(1, len(costs)):
        if values[i] * costs[m] > values[m] * costs[i]:
            m = i

    amount = min(costs[m], capacity)
    value = values[m] * amount / costs[m]

    costs.pop(m)
    values.pop(m)
    capacity -= amount

    return value + maximum_loot(capacity, costs, values)

if __name__ == "__main__":
    data = list(map(int, sys.stdin.read().split()))
    n, capacity = data[0:2]
    values = data[2:(2 * n + 2):2]
    weights = data[3:(2 * n + 2):2]
    opt_value = maximum_loot(capacity, weights, values)
    print(f'{opt_value:.10f}')
```

```
import sys

def maximum_loot(capacity, costs, values):
    value = 0.
    while capacity > 0:
        m = -1
        for i in range(len(costs)):
            if costs[i] > 0:
                if m == -1 or \
                    values[i] * costs[m] > \
                    values[m] * costs[i]:
                    m = i
        if m == -1:
            break
        amount = min(costs[m], capacity)
        value += values[m] * amount / costs[m]
        capacity -= amount
        costs[m] -= amount

    return value

if __name__ == "__main__":
    data = list(map(int, sys.stdin.read().split()))
    n, capacity = data[0:2]
    values = data[2:(2 * n + 2):2]
    weights = data[3:(2 * n + 2):2]
    opt_value = maximum_loot(capacity, weights, values)
    print(f'{opt_value:.10f}')
```

### 5.2.3 Car Fueling

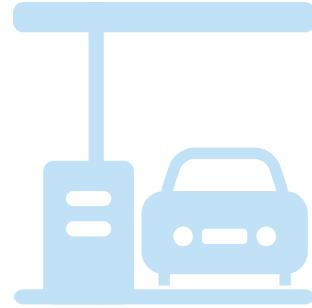
---

#### Car Fueling Problem

Compute the minimum number of gas tank refills to get from one city to another.

**Input:** Integers  $d$  and  $m$ , as well as a sequence of integers  $stop_1 < stop_2 < \dots < stop_n$ .

**Output:** The minimum number of refills to get from one city to another if a car can travel at most  $m$  miles on a full tank. The distance between the cities is  $d$  miles and there are gas stations at distances  $stop_1, stop_2, \dots, stop_n$  along the way. We assume that a car starts with a full tank.



---

Try our [Car Fueling](#) interactive puzzle before solving this programming challenge!

**Input format.** The first line contains an integer  $d$ . The second line contains an integer  $m$ . The third line specifies an integer  $n$ . Finally, the last line contains integers  $stop_1, stop_2, \dots, stop_n$ .

**Output format.** The minimum number of refills needed. If it is not possible to reach the destination, output  $-1$ .

**Constraints.**  $1 \leq d \leq 10^5$ .  $1 \leq m \leq 400$ .  $1 \leq n \leq 300$ .  $0 < stop_1 < stop_2 < \dots < stop_n < d$ .

**Sample 1.**

Input:

```
950
400
4
200 375 550 750
```

Output:

```
2
```

The distance between the cities is 950, the car can travel at most 400 miles on a full tank. It suffices to make two refills: at distance 375 and 750. This is the minimum number of refills as with a single refill one would only be able to travel at most 800 miles.

**Sample 2.**

Input:

```
10
3
4
1 2 5 9
```

Output:

```
-1
```

One cannot reach the gas station at point 9 as the previous gas station is too far away.

**Sample 3.**

Input:

```
200
250
2
100 150
```

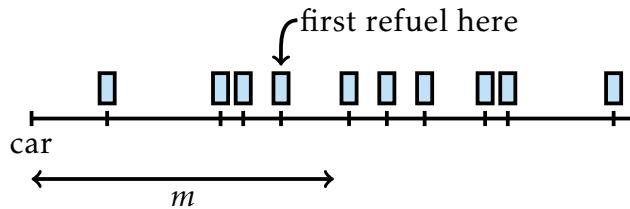
Output:

```
0
```

There is no need to refill the tank as the car starts with a full tank and can travel for 250 miles whereas the distance to the destination is 200 miles.

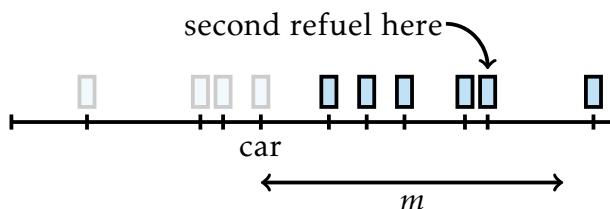
### Solution: Refuel at the Furthest Reachable Station

We are going to be greedy: when approaching a gas station, we only stop there if we cannot reach the next one without running out of gas. Pictorially, this looks as follows.



To prove that this strategy leads to an optimum solution, assume that  $stop_k$  is the furthest reachable station: on the one hand, it is reachable on a full tank ( $stop_k \leq m$ ), on the other hand, the next station is not reachable ( $stop_{k+1} > d$ ). We claim that there must be an optimum solution where the first refuel is at  $stop_k$ . To prove this, take an optimum solution and let  $stop_i$  be its first refuel. If  $i = k$ , then there is nothing to prove. Also,  $i$  cannot be larger than  $k$ , since  $stop_{k+1} > m$ . Hence  $i < k$ . Since  $stop_k$  is reachable on a full tank, we can safely move the first refuel to  $stop_k$ .

To design an algorithm, we just keep using the same idea: we move the car to  $stop_k$  and iterate.



The following recursive code exploits this idea and handles two important corner cases: when we cannot reach any gas station at all and when we reach the final destination. The variable *location* is the current location of the car that is equal to 0 initially.

```

REFILLS(location, Stops, m, d)
if location + m ≥ d:
    return 0
if Stops is empty or (Stops[0] – location) > m:
    return ∞
lastStop ← location
while Stops is not empty and (Stops[0] – location) ≤ m:
    lastStop ← Stops[0]
    pop 0-th element from Stops
return 1 + REFILLS(lastStop, Stops, m, d)

```

**Running time.** The running time is  $O(n)$ , since we just progress through the list of stops.

## Code

```

from sys import stdin

def compute_min_refills(distance, tank, stops):
    stops = [0] + stops + [distance]
    num_refills, cur_refill = 0, 0
    while cur_refill <= len(stops) - 2:
        last_refill = cur_refill
        while cur_refill <= len(stops) - 2 and \
            stops[cur_refill + 1] - stops[last_refill] <= tank:
            cur_refill += 1
        if cur_refill == last_refill:
            return -1
        if cur_refill <= len(stops) - 2:
            num_refills += 1

    return num_refills

if __name__ == '__main__':

```

```
d, m, _, *stops = map(int, stdin.read().split())
print(compute_min_refills(d, m, stops))
```

```
from sys import stdin

def refills(location, distance, tank, stops):
    if location + tank >= distance:
        return 0
    if not stops or stops[0] - location > tank:
        return float('inf')
    last_stop = location
    while stops and stops[0] - location <= tank:
        last_stop = stops[0]
        stops.pop(0)
    return 1 + refills(last_stop, distance, tank, stops)

if __name__ == '__main__':
    d, m, _, *stops = map(int, stdin.read().split())
    num_refills = refills(0, d, m, stops)
    print(-1 if num_refills == float('inf') else num_refills)
```

## 5.2.4 Maximum Advertisement Revenue

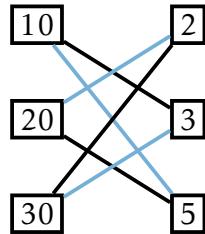
### Maximum Product of Two Sequences Problem

*Find the maximum dot product of two sequences of numbers.*

**Input:** Two sequences of  $n$  positive integers:  $price_1, \dots, price_n$  and  $clicks_1, \dots, clicks_n$ .

**Output:** The maximum value of  $price_1 \cdot c_1 + \dots + price_n \cdot c_n$ , where  $c_1, \dots, c_n$  is a permutation of  $clicks_1, \dots, clicks_n$ .

clicks      prices



You have  $n = 3$  advertisement slots on your popular Internet page and you want to sell them to advertisers. They expect, respectively,  $clicks_1 = 10$ ,  $clicks_2 = 20$ , and  $clicks_3 = 30$  clicks per day. You found three advertisers willing to pay  $price_1 = \$2$ ,  $price_2 = \$3$ , and  $price_3 = \$5$  per click. How would you pair the slots and advertisers to maximize the revenue? For example, the blue pairing shown above gives a revenue of  $10 \cdot 5 + 20 \cdot 2 + 30 \cdot 3 = 180$  dollars, while the black one results in revenue of  $10 \cdot 3 + 20 \cdot 5 + 30 \cdot 2 = 190$  dollars.

**Input format.** The first line contains an integer  $n$ , the second one contains a sequence of integers  $price_1, \dots, price_n$ , the third one contains a sequence of integers  $clicks_1, \dots, clicks_n$ .

**Output format.** Output the maximum value of  $(price_1 \cdot c_1 + \dots + price_n \cdot c_n)$ , where  $c_1, \dots, c_n$  is a permutation of  $clicks_1, \dots, clicks_n$ .

**Constraints.**  $1 \leq n \leq 10^3$ ;  $0 \leq price_i, clicks_i \leq 10^5$  for all  $1 \leq i \leq n$ .

**Sample 1.**

Input:

```
1  
23  
39
```

Output:

```
897  
897 = 23 · 39.
```

**Sample 2.**

Input:

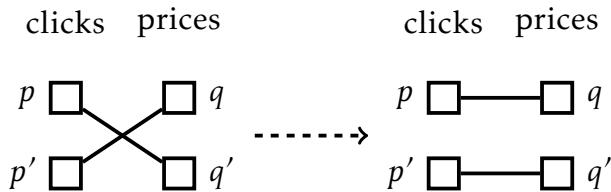
```
3  
2 3 9  
7 4 2
```

Output:

```
79  
79 = 7 · 9 + 2 · 2 + 3 · 4.
```

**Solution: Assign the Most Valuable Ad to the Most Popular Place**

You wouldn't be surprised to hear that a greedy strategy maximizes the revenue. Assume that  $\text{clicks}_p$  and  $\text{price}_q$  are the largest elements:  $\text{clicks}_p \geq \text{clicks}_i$  and  $\text{price}_q \geq \text{price}_i$  for all  $i = 1, \dots, n$ . We claim that there exists an optimum solution that pairs  $\text{clicks}_p$  with  $\text{price}_q$ . To prove this, consider an optimum solution and assume that it pairs  $\text{clicks}_p$  with  $\text{price}_{q'}$  for some  $q' \neq q$  and  $\text{price}_q$  with  $\text{clicks}_{p'}$  for some  $p' \neq p$ . We show that substituting pairs  $(p, q')$  and  $(p', q)$  with pairs  $(p, q)$  and  $(p', q')$  can only increase the total revenue:



Let's estimate how this substitution affects the total revenue. Before the substitution, the considered four elements contribute the following amount

to the revenue:

$$clicks_p \cdot price_{q'} + clicks_{p'} \cdot price_q.$$

After the substitution, they contribute the following amount:

$$clicks_p \cdot price_q + clicks_{p'} \cdot price_{q'}.$$

Hence, the substitution increases the total revenue by

$$\begin{aligned} &clicks_p \cdot price_q + clicks_{p'} \cdot price_{q'} - \\ &clicks_p \cdot price_{q'} - clicks_{p'} \cdot price_q = \\ &(clicks_p - clicks_{p'}) \cdot (clicks_q - clicks_{q'}) \geq 0. \end{aligned}$$

This leads to the algorithm that pairs the ad with the largest number of clicks with the largest price, removes them from further consideration, and iterates.

```
REVENUE(Click, Price):
revenue ← 0
while Clicks is not empty:
    p ← index with largest Click[p]
    q ← index with largest Price[q]
    revenue ← revenue + Clicks[p] · Price[q]
    remove p-th element of Click
    remove q-th element of Price
return revenue
```

**Running time.** The running time of this algorithm is  $O(n^2)$ : at each of  $n$  iterations, we perform two linear scans to find two largest elements. One can also sort the two lists in advance to avoid searching for the largest element at every iteration from scratch. This gives a solution with running time  $O(n \log n)$ .

## Code

```
from sys import stdin

def max_revenue(clicks, prices):
    revenue = 0
    while len(clicks) > 0:
        max_clicks = max(clicks)
        max_price = max(prices)
        revenue += max_clicks * max_price
        clicks.remove(max_clicks)
        prices.remove(max_price)
    return revenue

if __name__ == '__main__':
    input = stdin.read()
    data = list(map(int, input.split()))
    n = data[0]
    clicks = data[1:(n + 1)]
    prices = data[(n + 1):]
    print(max_revenue(clicks, prices))
```

```
from sys import stdin

def max_revenue(clicks, prices):
    clicks, prices = sorted(clicks), sorted(prices)
    return sum(clicks[i] * prices[i]
               for i in range(len(clicks)))

if __name__ == '__main__':
    input = stdin.read()
    data = list(map(int, input.split()))
    n = data[0]
```

```
clicks, prices = data[1:(n + 1)], data[(n + 1):]  
print(max_revenue(clicks, prices))
```

### 5.2.5 Collecting Signatures

---

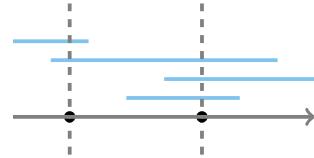
#### Covering Segments by Points Problem

*Find the minimum number of points needed to cover all given segments on a line.*

**Input:** A sequence of  $n$  segments  $[l_1, r_1], \dots, [l_n, r_n]$  on a line.

**Output:** A set of points of minimum size such that each segment  $[l_i, r_i]$  contains a point, i.e., there exists a point  $x$  from this set such that  $l_i \leq x \leq r_i$ .

---



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.

Try our [Touch All Segments](#) interactive puzzle before solving this programming challenge!

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

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

**Constraints.**  $1 \leq n \leq 100$ ;  $0 \leq l_i \leq r_i \leq 10^9$  for all  $i$ .

**Sample 1.**

Input:

```
3
1 3
2 5
3 6
```

Output:

```
1
3
```

All three segments  $[1, 3]$ ,  $[2, 5]$ ,  $[3, 6]$  contain the point with coordinate 3.

**Sample 2.**

Input:

```
4
4 7
1 3
2 5
5 6
```

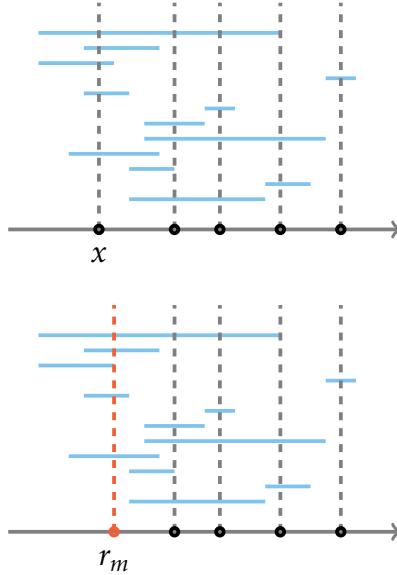
Output:

```
2
3 6
```

The second and the third segments contain the point with coordinate 3 while the first and the fourth segments contain the point with coordinate 6. All segments cannot be covered by a single point, since the segments  $[1, 3]$  and  $[5, 6]$  do not overlap. Another valid solution in this case is the set of points 2 and 5.

**Solution: Find a Segment with the Smallest Right End**

Consider the smallest ending point of a segment:  $r_m = \min\{r_1, \dots, r_n\}$ . We claim that there exists an optimum solution containing the point  $r_m$ . To prove this, take an optimum solution  $S$ . It must cover the segment  $[l_m, r_m]$ , hence  $S$  contains a point  $x$  such that  $l_m \leq x \leq r_m$ . If  $x = r_m$ , then we are done. Otherwise,  $x < r_m$ . In this case, we can replace  $x$  by  $r_m$  in  $S$ .



Clearly, this does not change the size of the solution  $S$ . To show that  $S$  is still a solution, assume, for the sake of contradiction, that some segment  $[l_i, r_i]$  is covered by  $x$ , but is not covered by  $r_m$ . This means that

$$l_i \leq x \leq r_i < r_m,$$

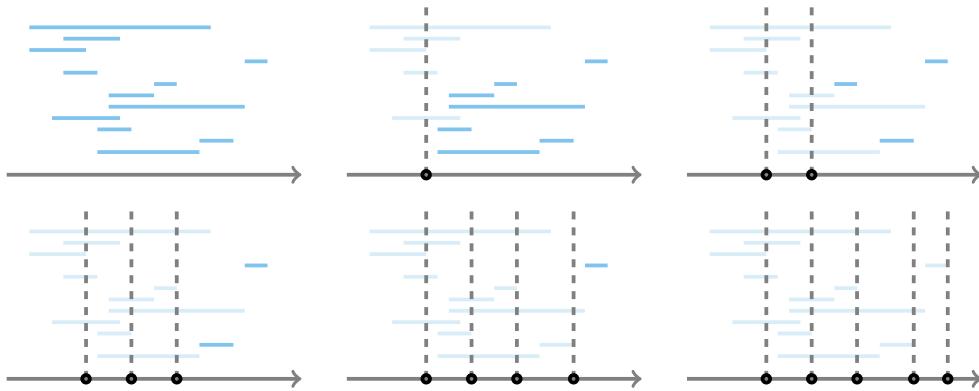
contradicting to the fact that  $r_m$  is the smallest right end.

This way, we arrive to the following algorithm: add to a solution the minimum right end  $r_m$ , discard all segments that are covered by  $r_m$ , and iterate.

```

SEGMENTS COVER(segments):
points ← empty set
while segments is not empty:
     $r_m \leftarrow$  minimum right endpoint of a segment from segments
    add  $r_m$  to points
    remove segments covered by  $r_m$  from the set segments
return points
    
```

The picture below shows an example.



The running time is  $O(n^2)$ , where  $n = |\text{segments}|$ , since there are at most  $n$  iterations of the `while` loop (at least one segment is discarded at each iteration) and each iteration boils down to two scans of the list `segments` (one scan to find the value of  $r$  and another one to remove segments that are covered by  $r$ ).

This algorithm is already sufficiently fast to pass the grader. To reduce the running time from  $O(n^2)$  to  $O(n \log n)$ , you can simply sort segments in increasing order of their right endpoints and scan the resulting list just once.

## Code

```
from sys import stdin
from collections import namedtuple

Segment = namedtuple('Segment', 'left right')

def is_covered(segment, point):
    return segment.left <= point <= segment.right

def segments_cover(segments):
    points = list()
    while len(segments) > 0:
        r = min([s.right for s in segments])
```

```
    points.append(r)
    segments = [s for s in segments
                if not is_covered(s, r)]
return points

if __name__ == '__main__':
    n, *data = map(int, stdin.read().split())
    input_segments = list(map(lambda x: Segment(x[0], x[1]),
                               zip(data[::2], data[1::2])))
    output_points = segments_cover(input_segments)
    print(len(output_points))
    print(*output_points)
```

```
output_points = segments_cover(input_segments)
print(len(output_points))
print(*output_points)
```

### 5.2.6 Maximum Number of Prizes

---

#### Distinct Summands Problem

Represent a positive integer as the sum of the maximum number of pairwise distinct positive integers.

8

**Input:** A positive integer  $n$ .

**Output:** The maximum  $k$  such that  $n$  can be represented as the sum  $a_1 + \dots + a_k$  of  $k$  distinct positive integers.

1 2 5

---

You are organizing a competition for children and have  $n$  candies to give as prizes. You would like to use these candies for top  $k$  places in this competition with a restriction that a higher place gets a larger number of candies. To make as many children happy as possible, you need to find the largest value of  $k$  for which it is possible.

Try our [Balls in Boxes](#) interactive puzzle before solving this programming challenge!

**Input format.** An integer  $n$ .

**Output format.** In the first line, output the maximum number  $k$  such that  $n$  can be represented as the sum of  $k$  pairwise distinct positive integers. In the second line, output  $k$  pairwise distinct positive integers that sum up to  $n$  (if there are multiple such representations, output any of them).

**Constraints.**  $1 \leq n \leq 10^9$ .

**Sample 1.**

Input:

6

Output:

3

1 2 3

### Sample 2.

Input:

8

Output:

3

1 2 5

### Sample 3.

Input:

2

Output:

1

2

### Solution

**Exercise Break.** Can one represent 8 as the sum of four positive distinct integers?

It is not difficult to see that the answer is negative. Indeed, assume that  $8 = a_1 + a_2 + a_3 + a_4$  and  $a_1 < a_2 < a_3 < a_4$ . Then,  $a_1 \geq 1$ ,  $a_2 \geq 2$ ,  $a_3 \geq 3$ , and  $a_4 \geq 4$ . But then  $a_1 + a_2 + a_3 + a_4 \geq 10$ .

For the same reason, if  $n$  equals the sum of  $k$  distinct positive integers  $a_1, \dots, a_k$ , then

$$n = a_1 + \dots + a_k \geq 1 + \dots + k = \frac{k(k+1)}{2}.$$

The converse is also true: if  $n \geq \frac{k(k+1)}{2}$ , then one can represent  $n$  as the sum of  $k$  distinct integers. Indeed, let  $\delta = n - \frac{k(k+1)}{2} > 0$ . Then,  $n$  is equal to the sum of the following integers:

$$1, 2, \dots, k-1, k+\delta.$$

The corresponding algorithm is straightforward: find the largest value of  $k$  such that  $\frac{k(k+1)}{2} \leq n$ . The running time is  $O(n)$ .

## Code

```
def get_prizes(n):
    k = 1
    while k * (k + 1) // 2 <= n:
        k += 1
    k -= 1

    delta = n - k * (k + 1) // 2
    return list(range(1, k)) + [k + delta]

if __name__ == '__main__':
    n = int(input())
    prizes = get_prizes(n)
    print(len(prizes))
    print(*prizes)
```

### 5.2.7 Maximum Salary

---

#### Largest Concatenate Problem

*Compile the largest number by concatenating the given numbers.*

**Input:** A sequence of positive integers.

**Output:** The largest number that can be obtained by concatenating the given integers in some order.



Resume

This is probably the most important problem in this book :). As the last question on an interview, your future boss gives you a few pieces of paper with a single number written on each of them and asks you to compose a largest number from these numbers. The resulting number is going to be your salary, so you are very motivated to solve this problem!

Try our [Largest Concatenate](#) interactive puzzle before solving this programming challenge!

Recall the algorithm for this problem that works for single digit numbers.

```
LARGESTCONCATENATE(Numbers):  
    yourSalary ← empty string  
    while Numbers is not empty:  
        maxNumber ←  $-\infty$   
        for each number in Numbers:  
            if number  $\geq$  maxNumber:  
                maxNumber ← number  
            append maxNumber to yourSalary  
            remove maxNumber from Numbers  
    return yourSalary
```

As we know already, this algorithm does not always maximize your salary: for example, for an input consisting of two integers 23 and 3 it returns 233, while the largest number is 323.

Not to worry, all you need to do to maximize your salary is to replace the line

```
if number ≥ maxNumber:
```

with the following line:

```
if IsBETTER(number, maxNumber):
```

for an appropriately implemented function IsBETTER. For example, IsBETTER(3, 23) should return True.

**Stop and Think.** How would you implement IsBETTER?

**Input format.** The first line of the input contains an integer  $n$ . The second line contains integers  $a_1, \dots, a_n$ .

**Output format.** The largest number that can be composed out of  $a_1, \dots, a_n$ .

**Constraints.**  $1 \leq n \leq 100$ ;  $1 \leq a_i \leq 10^3$  for all  $1 \leq i \leq n$ .

**Sample 1.**

Input:

```
2  
21 2
```

Output:

```
221
```

Note that in this case the above algorithm also returns an incorrect answer 212.

**Sample 2.**

Input:

```
5  
9 4 6 1 9
```

Output:

```
99641
```

The input consists of single-digit numbers only, so the algorithm above returns the correct answer.

### Sample 3.

Input:

```
3  
23 39 92
```

Output:

```
923923
```

The (incorrect) LARGESTNUMBER algorithm nevertheless produces the correct answer in this case, another reminder to always prove the correctness of your greedy algorithms!

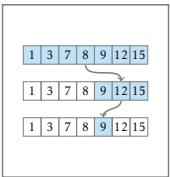
### Code

```
def largest_number(numbers):  
    for _ in numbers:  
        for i in range(len(numbers) - 1):  
            if numbers[i] + numbers[i + 1] < \  
                numbers[i + 1] + numbers[i]:  
                    t = numbers[i]  
                    numbers[i] = numbers[i + 1]  
                    numbers[i + 1] = t  
  
    return int(''.join(numbers))  
  
if __name__ == '__main__':  
    _ = int(input())  
    input_numbers = input().split()  
    print(largest_number(input_numbers))
```



# Chapter 6: Divide-and-Conquer

In this chapter, you will learn about divide-and-conquer algorithms that will help you to search huge databases a million times faster than brute-force algorithms. Armed with this algorithmic technique, you will learn that the standard way to multiply numbers (that you learned in the grade school) is far from being the fastest! We will then apply the divide-and-conquer technique to design fast sorting algorithms. You will learn that these algorithms are optimal, i.e., even the legendary computer scientist Alan Turing would not be able to design a faster sorting algorithm!

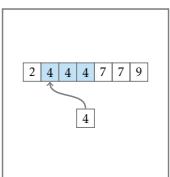


## Sorted Array Multiple Search

*Search multiple keys in a sorted sequence of keys.*

**Input.** A sorted array  $K$  of distinct integers and an array  $Q = \{q_0, \dots, q_{m-1}\}$  of integers.

**Output.** For each  $q_i$ , check whether it occurs in  $K$ .



## Binary Search with Duplicates

*Find the index of the first occurrence of a key in a sorted array.*

**Input.** A sorted array of integers (possibly with duplicates) and an integer  $q$ .

**Output.** Index of the first occurrence of  $q$  in the array or “ $-1$ ” if  $q$  does not appear in the array.

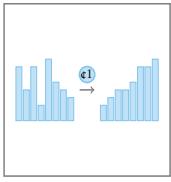


## Majority Element

*Check whether a given sequence of numbers contains an element that appears more than half of the times.*

**Input.** A sequence of  $n$  integers.

**Output.** 1, if there is an element that is repeated more than  $n/2$  times, and 0 otherwise.

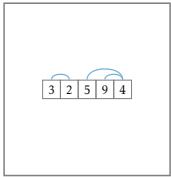


### Speeding-up RANDOMIZEDQUICKSORT

Sort a given sequence of numbers (that may contain duplicates) using a modification of RANDOMIZEDQUICKSORT that works in  $O(n \log n)$  expected time.

**Input.** An integer array with  $n$  elements that may contain duplicates.

**Output.** Sorted array (generated using a modification of RANDOMIZEDQUICKSORT) that works in  $O(n \log n)$  expected time.

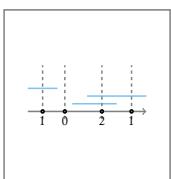


### Number of Inversions

Compute the number of inversions in a sequence of integers.

**Input.** A sequence of  $n$  integers  $a_1, \dots, a_n$ .

**Output.** The number of inversions in the sequence, i.e., the number of indices  $i < j$  such that  $a_i > a_j$ .

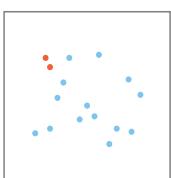


### Points and Segments

Given a set of points and a set of segments on a line, compute, for each point, the number of segments it is contained in.

**Input.** A list of  $n$  segments and a list of  $m$  points.

**Output.** The number of segments containing each point.



### Closest Points

Find the closest pair of points in a set of points on a plane.

**Input.** A list of  $n$  points on a plane.

**Output.** The minimum distance between a pair of these points.

## 6.1 The Main Idea

If you want to solve a problem using a divide-and-conquer strategy, you have to think about the following three steps:

1. Breaking a problem into smaller subproblems.
2. Solving each subproblem recursively.
3. Combining a solution to the original problem out of solutions to subproblems.

The first two steps is the “divide” part, whereas the last step is the “conquer” part. We illustrate this approach with a number of problems of progressing difficulty and then proceed to the programming challenges.

### 6.1.1 Guess a Number

**Interactive Puzzle “Guess a Number”.** In the “Guess a Number” game, your opponent has an integer  $1 \leq x \leq n$  in mind. You ask questions of the form “Is  $x = y$ ?” Your opponent replies either “yes”, or “ $x < y$ ” (that is, “my number is smaller than your guess”), or “ $x > y$ ” (that is, “my number is larger than your guess”). Your goal is to get the “yes” answer by asking the minimum number of questions. Let  $n = 3$ : your goal is to guess  $1 \leq x \leq 3$  by asking at most two questions. Can you do this? [Try it online \(level 1\)!](#)

If you ask “Is  $x = 1$ ?” and get the “yes” answer, then you are done. But what if the opponent replies “ $x > 1$ ”? You conclude that  $x$  is equal to either 2 or 3, but you only have one question left. Similarly, if you ask “Is  $x = 3$ ?", the opponent may reply “ $x < 3$ ” and you will not be able to get the desired “yes” response by asking just one more question.

Let’s see what happens if your first question is “Is  $x = 2$ ?” If the opponent replies that  $x = 2$ , then you are done. If she replies that  $x < 2$ , then you already know that  $x = 1$ . Hence, you just ask “Is  $x = 1$ ?” as your second question and get the desired “yes” response. If the opponent replies that  $x > 2$ , your next question “Is  $x = 3$ ?” will get the “yes” response.

**Exercise Break.** Guess an integer  $1 \leq x \leq 7$  by asking at most three questions. [Try it online \(level 2\)!](#)

You may have already guessed that you are going to start by asking “Is  $x = 4?$ ” The reason is that in both cases,  $x < 4$  and  $x > 4$ , we *reduce the size of the search space from 7 to 3* (and we already know how to solve the problem for 3 elements). :

- if  $x < 4$ , then  $x$  is equal to either 1, 2, or 3;
- if  $x > 4$ , then  $x$  is equal to either 5, 6, or 7.

This, in turn, means that in both cases you can invoke the solution to the previous problem. The resulting protocol of questions is shown in Figure 6.1.

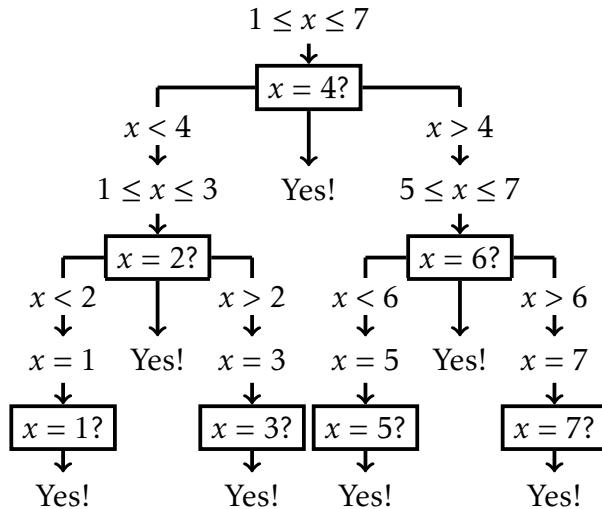


Figure 6.1: Guessing an integer  $1 \leq x \leq 7$  by asking at most three questions.

This strategy allows you to guess an integer  $1 \leq x \leq 2097151$  (over two million!) in just 21 questions. [Try it online \(level 4\)!](#)

The following Python code mimics the guessing process. The function `query` “knows” an integer  $x$ . A call to `query(y)` tells us whether  $x = y$ , or  $x > y$ , or  $x < y$ . The function `guess()` finds the number  $x$  by calling `query()`. It is called with two parameters, `lower` and `upper`, such that

$$\text{lower} \leq x \leq \text{upper},$$

that is,  $x$  lies in the segment  $[lower, upper]$ . It first computes the middle point of the segment  $[lower, upper]$  and then calls  $\text{query}(\text{middle})$ . If  $x < \text{middle}$ , then it continues with the interval  $[lower, \text{middle} - 1]$ . If  $x > \text{middle}$ , then it continues with the interval  $[\text{middle} + 1, upper]$ .

```
def query(y):
    x = 1618235
    if x == y:
        return 'equal'
    elif x < y:
        return 'smaller'
    else:
        return 'greater'

def guess(lower, upper):
    middle = (lower + upper) // 2
    answer = query(middle)
    print(f'Is x={middle}? It is {answer}.')
    if answer == 'equal':
        return
    elif answer == 'smaller':
        guess(lower, middle - 1)
    else:
        assert answer == 'greater'
        guess(middle + 1, upper)

guess(1, 2097151)
```

```
Is x=1048576? It is greater.
Is x=1572864? It is greater.
Is x=1835008? It is smaller.
Is x=1703936? It is smaller.
Is x=1638400? It is smaller.
Is x=1605632? It is greater.
```

```
Is x=1622016? It is smaller.  
Is x=1613824? It is greater.  
Is x=1617920? It is greater.  
Is x=1619968? It is smaller.  
Is x=1618944? It is smaller.  
Is x=1618432? It is smaller.  
Is x=1618176? It is greater.  
Is x=1618304? It is smaller.  
Is x=1618240? It is smaller.  
Is x=1618208? It is greater.  
Is x=1618224? It is greater.  
Is x=1618232? It is greater.  
Is x=1618236? It is smaller.  
Is x=1618234? It is greater.  
Is x=1618235? It is equal.
```

Try changing the value of  $x$  and run this code to see the sequence of questions (but make sure that  $x$  lies in the segment that `guess` is called with).

In general, our strategy for guessing an integer  $1 \leq x \leq n$  will require about  $\log_2 n$  questions. Recall that  $\log_2 n$  is equal to  $b$  if  $2^b = n$ . This means that if we keep dividing  $n$  by 2 until we get 1, we will make about  $\log_2 n$  divisions. What is important here is that  $\log_2 n$  is a *slowly growing* function: say, if  $n \leq 10^9$ , then  $\log_2 n < 30$ .

```
from math import log2  
  
def divide_till_one(n):  
    divisions = 0  
    while n > 1:  
        n = n // 2  
        divisions += 1  
    return divisions  
  
for d in range(1, 10):
```

```
n = 10 ** d
print(f'{n} {log2(n)} {divide_till_one(n)}')
```

```
10 3.321928094887362 3
100 6.643856189774724 6
1000 9.965784284662087 9
10000 13.287712379549449 13
100000 16.609640474436812 16
1000000 19.931568569324174 19
10000000 23.253496664211536 23
100000000 26.575424759098897 26
1000000000 29.897352853986263 29
```

### 6.1.2 Searching Sorted Data

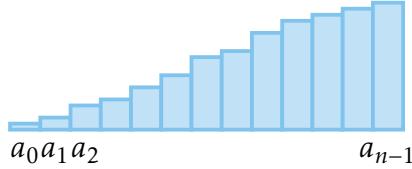
The method that we used for guessing a number is known as the *binary search*. Perhaps the most important application of binary search is *searching sorted data*. Searching is a fundamental problem: given a sequence and an element  $x$ , we would like to check whether  $x$  is present in this sequence. For example, 3 is present in the sequence (7, 2, 5, 6, 11, 3, 2, 9) and 4 is not present in this sequence. Given the importance of the search problem, it is not surprising that Python has built-in methods for solving it.

```
print(3 in [7, 2, 5, 6, 11, 3, 2, 9])
print(4 in [7, 2, 5, 6, 11, 3, 2, 9])
```

```
True
False
```

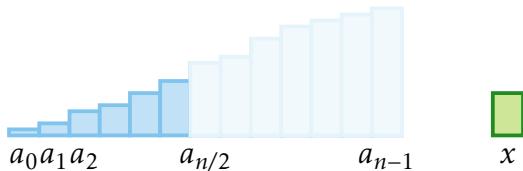
What is going on under the hood when one calls this `in` method? As you would expect, Python simply performs a *linear scan*. This linear scan makes up to  $n$  comparisons on a sequence of length  $n$ . If the sequence does not contain  $x$ , we *have to* scan all the elements: if we skip an element, we can't be sure that it is not equal to  $x$ .

Things change drastically if the given data is *sorted*, i.e., forms a sorted sequence  $a_0, \dots, a_{n-1}$  in increasing order.

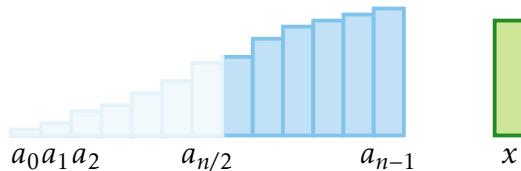


It turns out that in this case about  $\log_2 n$  comparisons are enough! This is a great speedup: the linear scan of a sorted array with a billion elements will take a billion comparisons, but binary search makes at most  $\log_2 10^9 < 30$  comparisons!

The idea is again to try to half the search space. To do this, we compare  $x$  with  $a_{n/2}$ . If  $x = a_{n/2}$ , then we are done. If  $x < a_{n/2}$ , then  $x$  can only appear in the first half of the array, implying that the right half can be discarded.



Similarly, if  $x > a_{n/2}$ , we discard the left half of the sequence as all its elements are certainly smaller than  $x$ :



This leads us to the following code.

```
def binary_search(a, x):
    print(f'Searching {x} in {a}')

    if len(a) == 0:
        return False

    if a[len(a) // 2] == x:
        print('Found!')
```

```

    return True
elif a[len(a) // 2] < x:
    return binary_search(a[len(a) // 2 + 1:], x)
else:
    return binary_search(a[:len(a) // 2], x)

```

```
binary_search([1, 2, 3, 3, 5, 6, 6, 8, 9, 9, 9], 8)
```

```

Searching 8 in [1, 2, 3, 3, 5, 6, 6, 8, 9, 9, 9]
Searching 8 in [6, 8, 9, 9, 9]
Searching 8 in [6, 8]
Found!

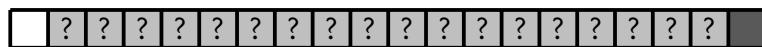
```

### 6.1.3 Finding a White-Black Pair

**Interactive Puzzle “White-Black Pair”.** Consider an array of white and black cells where the first cell is white and the last cell is black. A white cell followed by a black cell in this array is called a *white-black pair*. Here is an example of an array with six white-black pairs.



However, you don't see the colors of the cells (except for the first and the last one), for you, this array looks like this:



You may point to any cell and ask a question “What is its color?” Your goal is to find a white-black pair by asking the minimum number of questions. [Try it online!](#) After solving this puzzle, you will see that the binary search is useful even when the data is not sorted!

But how do we know that there is a white-black pair? This is true because when moving from the first white cell to the last black cell, the color of a cell should eventually switch from white to black at least once.

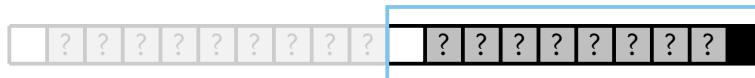
Even though this proof works for any array that starts from a white cell and ends in a black cell, it is *non-constructive*: it proves that there exists a white-black pair, but doesn't give an *efficient method* for finding this pair (by revealing the color of a few cells). In particular, if we start checking the color of the cells one by one from the left, then we will eventually find a white-black pair, but in the worst case it will require us to reveal the colors of all cells.

Thus, we know that a white-black pair exists, but we still need to figure out an efficient method for finding it. Inspired by our previous examples, let's reveal the color of the middle cell. Assume, for example, that it is white.



**Stop and Think.** If the middle cell turned out to be white, how would you proceed?

At first sight, it does not help us much. If we start revealing the color of its neighbor cells, both of them may be white. Instead, let's focus on the right part of the array.



Do you see? It is the same problem again! Since the leftmost cell is white, the rightmost one is black, it must contain a white-black pair. And its length is twice smaller.

If the middle cell turns out to be black, we can also find a subarray that starts in a white cell and ends in a black cell (though of length 11, but not 10).



Hence, in any case, we decrease the size by a factor of (almost) two: from a starting array of length  $n$ , we get an array of length  $\lceil \frac{n+1}{2} \rceil$ . Thus, after revealing the colors of at most five cells, we will find a white-black pair. Indeed, even in the worst possible case, the size of the array will shrink as follows:

$$20 \rightarrow 11 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 2.$$

When the size of the current sequence is reduced to two, we are left with an array consisting of just two cells, these two cells form a white-black since the first of them is white and the last is black.

#### 6.1.4 Finding a Peak

An element of a sequence is called a *peak* if it is greater than all its neighbors. Below, we highlight all peaks of a sequence. Note that the rightmost element is a peak since it is larger than its single neighbor.

3	4	2	12	13	5	8	9	7	6	1	10	15	17
---	---	---	----	----	---	---	---	---	---	---	----	----	----

**Interactive Puzzle “Finding a Peak”.** Consider an integer sequence with distinct unknown integers.

?	?	?	?	?	?	?	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Find (any) peak by revealing the minimum number of elements of this sequence. [Try it online!](#)

Note that any sequence (consisting of distinct integers) contains a peak: for example, the largest element is a peak. The largest element is a *global maximum* and it is not difficult to show that one needs to read the entire sequence to find it: if some element is not revealed, one cannot be sure that it is not the largest one. At the same time, a peak is a *local maximum* and the example above shows that a sequence may contain many local maxima. Below, we show that finding a peak can be done much faster than finding the global maximum.

Since every sequence has a peak element, we should simply find a peak in the left half and output it!

?	?	?	?	?	?	?	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---	---	---	---	---	---	---

**Exercise Break.** Find the flaw in this argument.

Indeed, as the example below illustrates, a peak in the left half, if it represents the last element in this half, is not necessarily a peak in the entire sequence.

1	2	5	6	7	3
---	---	---	---	---	---

Below, we denote the last element in the left half as *left* and the first element in the right half as *right*. If  $left > right$ , then we indeed can reduce search for a peak to searching the left half of the sequence only!

But how would we implement the divide and conquer if  $left < right$ ? In this case, we should find a peak in the right half and it will represent a peak for the entire sequence even if it happen to be the *right* element!

Thus, we start by revealing the values of two middle cells. Then, we recurse to the half of the sequence that contains the larger value (out of two revealed). This way, we guarantee that a peak found in this half is a peak in the orginal sequence. In the example below, one finds a peak by revealing six cells.

?	?	?	?	?	?	?	11	4	?	?	?	?	?	?	?	?
?	?	?	2	3	?	?	11	4	?	?	?	?	?	?	?	?
?	?	?	2	3	13	7	11	4	?	?	?	?	?	?	?	?

Since at each step we ask two questions and reduce the size of the sequence by a factor of two, the total number of questions is at most  $2 \log_2 n$ .

### 6.1.5 Multiplying Integers

If you are asked to multiply 25 and 63 and you don't have a calculator handy, what would you do? You will probably use the multiplication algorithm you learned in the elementary school, multiplying each digit from one number with each digit from the other and then adding up the products:

$$\begin{array}{r}
 & 2 & 5 \\
 \times & 6 & 3 \\
 \hline
 & 1 & 5 \\
 & 6 & 0 \\
 + & 3 & 0 & 0 \\
 \hline
 1 & 2 & 0 & 0 \\
 \hline
 1 & 5 & 7 & 5
 \end{array}$$

**Stop and Think.** What is the running time of this algorithm applied to two  $n$ -digit numbers? Can you propose a faster multiplication algorithm?

Since this algorithm multiplies each digit of the first number with each digit of the second number, it requires the quadratic number of multiplications. Can it be done faster, e.g., in  $O(n^{1.5})$ ?

In 1960, the great Russian mathematician Andrey Kolmogorov conjectured that the multiplication algorithm you learned in the elementary school is asymptotically optimal and thus cannot be improved in the sense of the big- $O$  notation. Within a week after he presented this conjecture at a seminar, a 23-year-old student Anatoly Karatsuba proved him wrong! Below we describe the Karatsuba algorithm.

**A different (more complex, but fast!) multiplication algorithm.** Let's multiply two-digit numbers  $AB$  and  $CD$ , where  $AB$  is  $10A + B$  and  $CD$  is  $10C + D$ :

$$(10A + B) \times (10C + D) = 100(A \times C) + 10(A \times D) + 10(B \times C) + B \times D.$$

Here, we have four small multiplications  $A \times C$ ,  $A \times D$ ,  $B \times C$ , and  $B \times D$  and it appears it cannot be done with fewer multiplications... Note that we do not count multiplications by 100 and 10 as “real” multiplications as they amount to simply adding zeroes.

**Exercise Break.** Can you multiply  $AB$  and  $CD$  with three multiplications using the hint below?

$$(10A + B) \times (10C + D) = \textcolor{blue}{100(A \times C)} + \textcolor{red}{10(A \times D) + 10(B \times C)} + \textcolor{green}{B \times D}.$$

Karatsuba noticed that all four multiplications in the formula above are required when we compute the product of  $A + B$  and  $C + D$ :

$$(A + B) \times (C + D) = A \times C + A \times D + B \times C + B \times D.$$

At this point, you probably wonder why we care about  $(A + B) \times (C + D)$  when our goal is to compute  $(10A + B) \times (10C + D)$ ... The Karatsuba algorithm is based on an observation that the equation above implies:

$$(A + B) \times (C + D) - A \times C - B \times D = A \times D + B \times C. \quad (6.1)$$

Thus, one can start by computing  $(A + B) \times (C + D)$ ,  $A \times C$ , and  $B \times D$  with only **three** multiplications and use the equation above to compute  $A \times D + B \times C$  without additional multiplications! Afterward, we can compute

$$(10A + B) \times (10C + D) = 100(A \times C) + 10(A \times D + B \times C) + B \times D.$$

with only three multiplications as:

$$100(A \times C) + 10[(A + B) \times (C + D) - A \times C - B \times D] + B \times D.$$

**Extending Karatsuba's idea from two-digit integers to arbitrary integers.** Karatsuba noticed that the trick with saving one multiplication for multiplying two-digit numbers works for any integers. Indeed, an  $n$ -digit integer can be split into two parts, each part with  $n/2$  digits. For example, if  $X = 54192143$  and  $Y = 885274$ , we can split  $X$  into parts  $A = 5419$  and  $B = 2143$  and split  $Y$  into parts  $C = 0088$  and  $D = 5274$  (note that we added two leading zeroes to  $B$  so that  $A$  and  $B$  have the same length).

Now,  $X = 10^{n/2}A + B$  and  $Y = 10^{n/2}C + D$ . For our working example,

$$\begin{aligned} X &= 54192143 = 54190000 + 2143 = 10^4A + B, \\ Y &= 885274 = 880000 + 5274 = 10^4C + D. \end{aligned}$$

Then,

$$\begin{aligned} X \times Y &= (10^{n/2}A + B) \times (10^{n/2}C + D) = \\ &= 10^n(A \times C) + 10^{n/2}(A \times D + B \times C) + B \times D. \end{aligned}$$

This way, we reduce multiplication of a pair of  $n$ -digit integers to multiplication of four pairs of  $n/2$ -digit integers. When the four products

$$A \times C, \quad A \times D, \quad B \times C, \quad B \times D$$

are already computed, it remains to multiply two times by a power of ten and to add three numbers. Both these operations take linear time; to multiply by a power of ten, one needs to append a number of zeros; adding two numbers can be done in linear time by a straightforward algorithm.

Since the identity (6.1) works for any integers  $A, B, C$ , and  $D$ , it allows one to save one multiplication of two  $n/2$ -digit integers:

$$X \times Y = 10^n(A \times C) + 10^{n/2}[(A + B) \times (C + D) - A \times C - B \times D] + B \times D.$$

This formula results in the following recursive multiplication algorithm.

```
KARATSUBA( $X, Y$ ):
 $n \leftarrow$  length of the largest of  $X$  and  $Y$ 
if  $n \leq 1$ :
    return  $X \times Y$ 
represent  $X$  as  $10^{n/2}A + B$  and  $Y$  as  $10^{n/2}C + D$ 
 $P \leftarrow$  KARATSUBA( $A, C$ )
 $Q \leftarrow$  KARATSUBA( $A + B, C + D$ )
 $R \leftarrow$  KARATSUBA( $B, D$ )
return  $10^n P + 10^{n/2}(Q - P - R) + R$ 
```

The running time  $T(n)$  of this algorithm satisfies a recurrence

$$T(n) \leq 3T(n/2) + O(n).$$

Indeed, it makes three recursive calls for  $n/2$ -digit integers.<sup>1</sup> Everything else (splitting  $X$  and  $Y$  into  $A, B, C$ , and  $D$ , multiplying by  $10^n$  and  $10^{n/2}$ , adding, and subtracting) is performed in linear time. We will soon learn that this recurrence implies that

$$T(n) \leq O(n^{\log_2 3}) = O(n^{1.584\dots}).$$

---

<sup>1</sup>We are cheating a bit: the integers  $A + B$  and  $C + D$  may have  $n/2 + 1$  digits. Still, the recurrence  $T(n) \leq 3T(n/2 + 1) + O(n)$  implies the same bound.

### 6.1.6 The Master Theorem

A typical divide-and-conquer algorithm solves a problem of size  $n$  as follows.

**Divide:** break the problem into  $a$  problems of size at most  $n/b$  and solve them recursively. We assume that  $a \geq 1$  and  $b > 1$ .

**Conquer:** combine the answers found by recursive calls to an answer for the initial problem.

Thus, the algorithm makes  $a$  recursive calls to problems of size  $n/b$ .<sup>2</sup> Assume that everything that is done outside of recursive calls takes time  $O(n^d)$  where  $d$  is non-negative constant. This includes preparing the recursive calls and combining the results.

Since this is a recursive algorithm, one also needs to specify conditions under which a problem is solved directly rather than recursively (with no such base case, a recursion would never stop). This usually happens when  $n$  becomes small. For simplicity, we assume that it is  $n = 1$  and that the running time of the algorithm is equal to 1 in this case.

Thus, by denoting the running time of the algorithm on problems of size  $n$  by  $T(n)$ , we get the following *recurrence relation* on  $T(n)$ :

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ aT\left(\frac{n}{b}\right) + O(n^d) & \text{if } n > 1. \end{cases}$$

Below, we show that one can find out the growth rate of  $T(n)$  by looking at parameters  $a$ ,  $b$ , and  $d$ . Before doing this, we illustrate how this recurrence relation captures some well-known algorithms.

**Binary search.** To search for a key in a sorted sequence of length  $n$ , the binary search algorithm makes a single recursive call for a problem of size  $n/2$ . Outside this recursive call it spends time  $O(1)$ . Thus,  $a = 1, b = 2, d = 0$ . Therefore,

$$T(n) = T\left(\frac{n}{2}\right) + O(1).$$

---

<sup>2</sup>Note that  $b$  does not necessarily divide  $n$ , so instead of  $n/b$  one should usually write  $\lceil n/b \rceil$ . Still, we write  $n/b$ : it simplifies the notation and does not break the analysis.

The running time of the binary search algorithm is  $O(\log n)$  since there are at most  $\log_2 n$  recursive calls. Another way to see this is to “unwind” the recurrence relation as shown below, that is, to apply the same equality to  $T(n/2)$ ,  $T(n/4)$ , and so on:

$$\begin{aligned}
T(n) &= T(n/2) + c = \\
&= T(n/4) + 2c = \\
&= T(n/8) + 3c = \\
&\vdots \\
&= T(n/2^k) + kc = \\
&\vdots \\
&= T(1) + \log_2 n \cdot c = O(\log n)
\end{aligned}$$

(here,  $c$  is a constant from the  $O(1)$  term).

**Merge sort.** To sort an array of length  $n$ , the MERGESORT algorithm breaks it into two subarrays of size  $n/2$ , sorts them recursively, and merges the results. The time spent by the algorithm before and after two recursive calls is  $O(n)$ . Thus,  $a = 2, b = 2, d = 1$ . Therefore,

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

Unwinding gives  $T(n) = O(n \log n)$ :

$$\begin{aligned}
T(n) &= 2T(n/2) + cn = \\
&= 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn = \\
&= 4(2T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn = \\
&\vdots \\
&= 2^k T(n/2^k) + kcn = \\
&\vdots \\
&= nT(1) + \log_2 n \cdot cn = O(n \log n).
\end{aligned}$$

**Integer multiplication.** To multiply two  $n$ -digit numbers, the Karatsuba algorithm breaks each number into two numbers with  $n/2$  digits. The

algorithm then computes the sum of some of these four numbers and makes three recursive calls. Finally, it computes the answer for the original problem based on the results of these recursive calls. Preparing the recursive calls and combining the results amounts to computing a few sums and hence takes time  $O(n)$ . Thus,  $a = 3, b = 2, d = 1$ . Therefore,

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n).$$

We are ready to state a theorem that allows one to determine the order of growth of  $T(n)$  by simply looking at parameters  $a, b$ , and  $d$ .

**Master Theorem.** *If  $T(n) = aT(n/b) + O(n^d)$  for constants  $a > 0, b > 1$ , and  $d \geq 0$ , then*

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } \log_b a > d, \\ O(n^d \log n) & \text{if } \log_b a = d, \\ O(n^d) & \text{if } \log_b a < d, \end{cases}$$

Thus, to determine the running time of the algorithm, one compares  $\log_b a$  and  $d$ . Note that this is the same as comparing  $a$  and  $b^d$ .

Before proving the theorem, we prove a technical lemma. It sheds some light on where the three cases in the Master Theorem come from. Recall that a sequence  $(1, \alpha, \alpha^2, \dots, \alpha^n)$  is called a *geometric progression* with ratio  $\alpha$ . Its sum

$$\sum_{i=0}^n \alpha^i = 1 + \alpha + \dots + \alpha^n$$

is known as the *geometric series*.

**Lemma 6.1.1** (order of growth of the geometric series). *Let  $\alpha$  be a positive constant. Then,*

$$\sum_{i=0}^n \alpha^i = \begin{cases} \Theta(\alpha^n) & \text{if } \alpha > 1, \\ \Theta(n) & \text{if } \alpha = 1, \\ \Theta(1) & \text{if } \alpha < 1. \end{cases} \quad (6.2)$$

*That is, if geometric progression is increasing, then the series grows as its last (and largest) term. If it is constant, then it grows as the number of terms. Finally, if it is decreasing, it is bounded by a constant.*

*Proof.* A convenient thing about geometric series is that after multiplying it by  $(\alpha - 1)$ , almost everything cancels out (or telescopes):

$$\begin{aligned} (1 + \alpha + \alpha^2 + \cdots + \alpha^n)(\alpha - 1) &= \\ &= \alpha + \alpha^2 + \cdots + \alpha^n + \alpha^{n+1} - \\ &- 1 - \alpha - \alpha^2 - \cdots - \alpha^n = \\ &= \alpha^{n+1} - 1 \end{aligned}$$

When  $\alpha \neq 1$ , one can divide by  $(\alpha - 1)$  to get a closed form expression for the geometric series:

$$\sum_{i=0}^n \alpha^i = \frac{\alpha^{n+1} - 1}{\alpha - 1} \quad (6.3)$$

Now, consider three cases.

1.  $\alpha > 1$ . Then,  $\alpha^n(\alpha - 1) < \alpha^{n+1} - 1 < \alpha^{n+1}$  and (6.3) can be bounded as follows:

$$\alpha^n < \frac{\alpha^{n+1} - 1}{\alpha - 1} < \frac{\alpha}{\alpha - 1} \alpha^n.$$

2.  $\alpha = 1$ . Then, we cannot use the formula (6.3), but the series is equal to  $n + 1$ .

3.  $\alpha < 1$ . Then,  $1 - \alpha < 1 - \alpha^{n+1} < 1$  and (6.3) can be bounded as follows:

$$1 < \frac{1 - \alpha^{n+1}}{1 - \alpha} < \frac{1}{1 - \alpha}.$$

□

*Proof of the Master Theorem.* One way to estimate  $T(n)$  is to unwind it. Denote by  $c$  the constant hidden in  $O(n^d)$ : on a problem of size  $n$ , the algorithm spends time at most  $cn^d$  for preparing the recursive calls and combining

their results.

$$\begin{aligned}
T(n) &= cn^d + aT(n/b) \\
&= cn^d + a(c(n/b)^d + aT(n/b^2)) = cn^d + ca(n/b)^d + a^2T(n/b^2) = \\
&\vdots \\
&= cn^d + ca(n/b)^d + ca^2(n/b^2)^d + \dots = \\
&= \sum_{l=0}^{\log_b n} ca^l \left(\frac{n}{b^l}\right)^d = \tag{6.4}
\end{aligned}$$

$$\begin{aligned}
&= \sum_{l=0}^{\log_b n} a^l \cdot c \cdot \frac{n^d}{b^{ld}} = \\
&= cn^d \sum_{l=0}^{\log_b n} \frac{a^l}{b^{ld}} = \\
&= cn^d \sum_{l=0}^{\log_b n} \left(\frac{a}{b^d}\right)^l = \\
&= cn^d \sum_{l=0}^{\log_b n} \alpha^l, \text{ where } \alpha = \frac{a}{b^d}. \tag{6.5}
\end{aligned}$$

Below, we show another way of arriving at the same sum and then estimate the order of growth of the sum.

To solve a problem of size  $n$ , the algorithm makes  $a$  recursive calls to problems of size  $n/b$ . To solve each of the problems of size  $n/b$ , the algorithm, again, makes  $a$  recursive calls to problems of size  $n/b^2$  and so on. This gives rise to a recursion tree shown in Figure 6.2 where the root has level 0, its children have level 1, and so on. Since by going one level down, we reduce the problem size by  $b$ , the total number of levels is  $\log_b n$ . The number of problems at level  $l$  is equal to  $a^l$  and the size of each problem is  $\frac{n}{b^l}$ .

Now, let us estimate the total work the algorithm performs at level  $l$  outside of recursive calls. Since there are  $a^l$  problems of size  $n/b^l$  at level  $l$ , the total work performed at this level is

$$a^l \cdot c \cdot \left(\frac{n}{b^l}\right)^d.$$

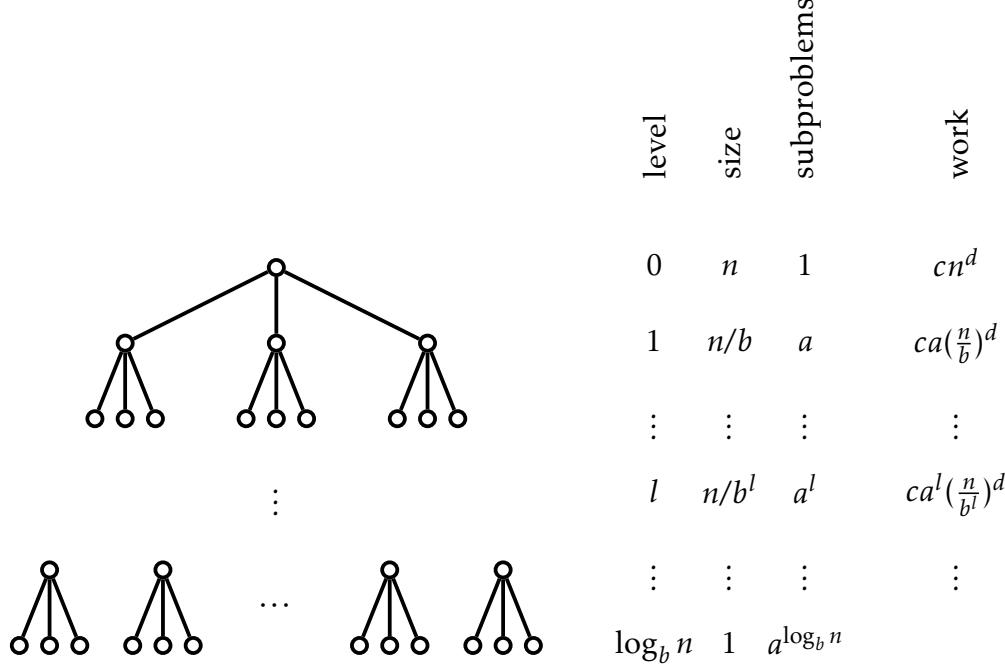


Figure 6.2: A tree of recursive calls of a divide-and-conquer algorithm for  $a = 3$ .

By taking the sum over all levels  $l = 0, 1, \dots, \log_b n$ , we get (6.4). Thus, it remains to estimate the order of growth of (6.5).

The sum (6.5) is a geometric series with ratio  $\alpha = \frac{a}{b^d}$ . Its growth rate depends on whether  $a$  is smaller, equal, or larger than  $b^d$  (recall (6.2)). Consider these three cases.

1.  $a > b^d$ . Then, the geometric progression is increasing and the series grows as its last term:

$$cn^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n} = cn^d \cdot \frac{a^{\log_b n}}{b^{d \log_b n}} = cn^d \cdot \frac{n^{\log_b a}}{n^d} = c \cdot n^{\log_b a} = O(n^{\log_b a}).$$

(We used the identities  $a^{\log_b n} = n^{\log_b a}$  and  $b^{\log_b n} = n$ .)

2.  $a = b^d$ . Then, the series grows as its number of terms:

$$cn^d \cdot O(\log_b n) = O(n^d \log n).$$

(Recall that one can omit the base of a logarithm inside big- $O$  if the base is constant.)

3.  $a < b^d$ . Then, the geometric progression is decreasing and the series grows as a constant:

$$cn^d \cdot O(1) = O(n^d).$$

□

**Exercise Break.** Find the order of growth of  $T(n)$  if it satisfies the following recurrences:

- $T(n) \leq 5T(n/4) + O(n)$
- $T(n) \leq 5T(n/4) + O(n^2)$
- $T(n) \leq 7T(n/2) + O(n^2)$

To conclude, let us show a few recurrences that are not covered (at least, directly) by the Master Theorem.

- $T(n) \leq T(n/3) + T(2n/3) + O(n)$ . In this case, there are two recursive calls, but for problems of *different* size. In particular, the tree of recursive calls is going to be imbalanced. Still, by essentially the same analysis one can show that  $T(n) \leq O(n \log n)$ : though the tree is imbalanced, its depth is logarithmic, and the work at every level is still  $O(n)$ .
- $T(n) \leq 2T(n - 1) + O(1)$ . In this case, the size of subproblems is not of the form  $n/b$ . It is possible to show that  $T(n) \leq O(2^n)$  in this case: either by induction or by unwinding the recurrence. In particular, if one drops the  $O(1)$  term (and assumes, as usual, that  $T(1) = 1$ ), then  $T(n)$  is exactly the the number of leaves in a binary tree of depth  $n$ , that is,  $2^n$ .
- $T(n) = T(\sqrt{n}) + O(1)$ . Again, the size of a subproblem in this case is not of the form  $n/b$ . One can write  $n = 2^k$  and then rewrite the recurrence as  $T'(k) = T'(k/2) + O(1)$ . The Master Theorem then implies that  $T'(k) = O(\log k)$  and hence  $T(n) = O(\log \log n)$ . Another way to see this is to unwind the recurrence.

## 6.2 Programming Challenges

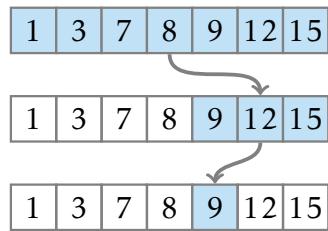
### 6.2.1 Binary Search

#### Sorted Array Search Problem

Search a key in a sorted array of keys.

**Input:** A sorted array  $K = [k_0, \dots, k_{n-1}]$  of distinct integers (i.e.,  $k_0 < k_1 < \dots < k_{n-1}$ ) and an integer  $q$ .

**Output:** Check whether  $q$  occurs in  $K$ .



A naive way to solve this problem, is to scan the array  $K$  (running time  $O(n)$ ). The `BINARYSEARCH` algorithm below solves the problem in  $O(\log n)$  time. It is initialized by setting  $\text{minIndex}$  equal to 0 and  $\text{maxIndex}$  equal to  $n-1$ . It sets  $\text{midIndex}$  to  $(\text{minIndex}+\text{maxIndex})/2$  and then checks to see whether  $q$  is greater than or less than  $K[\text{midIndex}]$ . If  $q$  is larger than this value, then `BINARYSEARCH` iterates on the subarray of  $K$  from  $\text{minIndex}$  to  $\text{midIndex}-1$ ; otherwise, it iterates on the subarray of  $K$  from  $\text{midIndex}+1$  to  $\text{maxIndex}$ . Iteration eventually identifies whether  $q$  occurs in  $K$ .

```
BINARYSEARCH( $K[0..n-1], q$ )
minIndex  $\leftarrow 0$ 
maxIndex  $\leftarrow n-1$ 
while maxIndex  $\geq$  minIndex:
    midIndex  $\leftarrow \lfloor (\text{minIndex} + \text{maxIndex})/2 \rfloor$ 
    if  $K[\text{midIndex}] = q$ :
        return midIndex
    else if  $K[\text{midIndex}] < q$ :
        minIndex  $\leftarrow \text{midIndex} + 1$ 
    else:
        maxIndex  $\leftarrow \text{midIndex} - 1$ 
return -1
```

For example, if  $q = 9$  and  $K = [1, 3, 7, 8, 9, 12, 15]$ , `BINARYSEARCH` would

first set  $\text{minIndex} = 0$ ,  $\text{maxIndex} = 6$ , and  $\text{midIndex} = 3$ . Since  $q$  is greater than  $K[\text{midIndex}] = 8$ , we examine the subarray whose elements are greater than  $K[\text{midIndex}]$  by setting  $\text{minIndex} = 4$ , so that  $\text{midIndex}$  is recomputed as  $(4 + 6)/2 = 5$ . This time,  $q$  is smaller than  $K[\text{midIndex}] = 12$ , and so we examine the subarray whose elements are smaller than this value. This subarray consists of a single element, which is  $q$ .

The running time of `BINARYSEARCH` is  $O(\log n)$  since it reduces the length of the subarray by at least a factor of 2 at each iteration of the `while` loop. Note however that our grading system is unable to check whether you implemented a fast  $O(\log n)$  algorithm for the Sorted Array Search or a naive  $O(n)$  algorithm. The reason is that any program needs a linear time in order to just read the input data. For this reason, we ask you to solve the following more general problem.

### Sorted Array Multiple Search Problem

*Search multiple keys in a sorted sequence of keys.*

**Input:** A sorted array  $K$  of distinct integers and an array  $Q = [q_0, \dots, q_{m-1}]$  of integers.

**Output:** For each  $q_i$ , check whether it occurs in  $K$ .

**Input format.** The first two lines of the input contain an integer  $n$  and a sequence  $k_0 < k_1 < \dots < k_{n-1}$  of  $n$  distinct positive integers in increasing order. The next two lines contain an integer  $m$  and  $m$  positive integers  $q_0, q_1, \dots, q_{m-1}$ .

**Output format.** For all  $i$  from 0 to  $m-1$ , output an index  $0 \leq j \leq n-1$  such that  $k_j = q_i$ , or  $-1$ , if there is no such index.

**Constraints.**  $1 \leq n \leq 3 \cdot 10^4$ ;  $1 \leq m \leq 10^5$ ;  $1 \leq k_i \leq 10^9$  for all  $0 \leq i < n$ ;  $1 \leq q_j \leq 10^9$  for all  $0 \leq j < m$ .

## Sample.

Input:

```
5
1 5 8 12 13
5
8 1 23 1 11
```

Output:

```
2 0 -1 0 -1
```

Queries 8, 1, and 1 occur at positions 3, 0, and 0, respectively, while queries 23 and 11 do not occur in the sequence of keys.

**Programming tips:** do not use built-in binary search ([LC6](#)).

## Solution

```
def binary_search(keys, query):
    min_index, max_index = 0, len(keys) - 1
    while max_index >= min_index:
        mid_index = (min_index + max_index) // 2
        if keys[mid_index] == query:
            return mid_index
        elif keys[mid_index] < query:
            min_index = mid_index + 1
        else:
            max_index = mid_index - 1
    return -1

if __name__ == '__main__':
    num_keys = int(input())
    input_keys = list(map(int, input().split()))
    assert len(input_keys) == num_keys

    num_queries = int(input())
    input_queries = list(map(int, input().split()))
    assert len(input_queries) == num_queries
```

```
for q in input_queries:  
    print(binary_search(input_keys, q), end=' ')
```

## 6.2.2 Binary Search with Duplicates

Donald Knuth, the author of *The Art of Computer Programming*, famously said: “Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky.” He was referring to a modified classical Binary Search Problem:

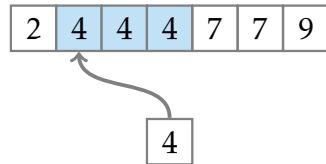
---

### Binary Search with Duplicates Problem

*Find the index of the first occurrence of a key in a sorted array.*

**Input:** A sorted array of integers (possibly with duplicates) and an integer  $q$ .

**Output:** Index of the first occurrence of  $q$  in the array or “ $-1$ ” if  $q$  does not appear in the array.



When Knuth asked professional programmers at top companies like IBM to implement an efficient algorithm for binary search with duplicates, 90% of them had bugs — year after year. Indeed, although the binary search algorithm was first published in 1946, the first bug-free algorithm for binary search with duplicates was published only in 1962!

Similarly to the previous problem, here we ask you to search for  $m$  integers rather than a single one.

**Input format.** The first two lines of the input contain an integer  $n$  and a sequence  $k_0 \leq k_1 \leq \dots \leq k_{n-1}$  of  $n$  positive integers in non-decreasing order. The next two lines contain an integer  $m$  and  $m$  positive integers  $q_0, q_1, \dots, q_{m-1}$ .

**Output format.** For all  $i$  from 0 to  $m - 1$ , output the index  $0 \leq j \leq n - 1$  of the first occurrence of  $q_i$  (i.e.,  $k_j = q_i$ ) or  $-1$ , if there is no such index.

**Constraints.**  $1 \leq n \leq 3 \cdot 10^4$ ;  $1 \leq m \leq 10^5$ ;  $1 \leq k_i \leq 10^9$  for all  $0 \leq i < n$ ;  $1 \leq q_j \leq 10^9$  for all  $0 \leq j < m$ .

### Sample.

Input:

```
7  
2 4 4 4 7 7 9  
4  
9 4 5 2
```

Output:

```
6 1 -1 0
```

**Programming tips:** do not use built-in binary search ([LC6](#)).

### Solution

Given a key  $q$ , your goal is to find the first (top) occurrence of this key in the array  $K$ . For example, if  $K = \{3, 6, 6, 7, 7, 7, 7, 9\}$  and the key  $q$  is 7, then the first occurrence of this key is located at index 3. Of course, you can find one of the occurrences of the key by simply launching the binary search. Afterward, you can find the first occurrence of the key by consecutively checking the element before the position of the found element as illustrated by the blue lines in the pseudocode below.

```
NAIVEBINARYSEARCHWITHDUPLICATES( $K[0..n-1], q$ )  
 $minIndex \leftarrow 0$   
 $maxIndex \leftarrow n-1$   
while  $maxIndex \geq minIndex$ :  
     $midIndex \leftarrow \lfloor (minIndex + maxIndex)/2 \rfloor$   
    if  $K[midIndex] = q$ :  
        top  $\leftarrow midIndex$   
        while  $top > 0$  and  $K[top-1] = K[top]$ :  
            top  $\leftarrow top - 1$   
        return top  
    else if  $K[midIndex] < q$ :  
        minIndex  $\leftarrow midIndex + 1$   
    else:  
        maxIndex  $\leftarrow midIndex - 1$   
return -1
```

**Stop and Think.** What is the running time of this algorithm?

This algorithm may become slow in the case of an array with many duplicates. For example, if a single duplicated element accounts for half of an array, `NAIVEBINARYSEARCHWITHDUPLICATES` takes linear  $O(n)$  time instead of logarithmic  $O(\log n)$  time. This problem is fixed in the pseudocode below.

```
BINARYSEARCHWITHDUPLICATES( $K[0..n - 1], q$ )
minIndex  $\leftarrow 0$ 
maxIndex  $\leftarrow n - 1$ 
result  $\leftarrow -1$ 
while maxIndex  $\geq$  minIndex:
    midIndex  $\leftarrow \lfloor (minIndex + maxIndex)/2 \rfloor$ 
    if  $K[midIndex] = q$ :
        maxIndex  $\leftarrow midIndex - 1$ 
        result  $\leftarrow midIndex$ 
    else if  $K[midIndex] < q$ :
        minIndex  $\leftarrow midIndex + 1$ 
    else:
        maxIndex  $\leftarrow midIndex - 1$ 
return result
```

## Code

Below, we provide two implementations: the first one directly implements the pseudocode discussed above, the second one uses the built-in `bisect_left` method.

```
def binary_search(keys, query):
    min_index, max_index = 0, len(keys) - 1
    result = -1
    while max_index  $\geq$  min_index:
        mid_index = (min_index + max_index) // 2
        if keys[mid_index] == query:
            max_index = mid_index - 1
            result = mid_index
        elif keys[mid_index] < query:
            min_index = mid_index + 1
        else:
```

```

        max_index = mid_index - 1
    return result

if __name__ == '__main__':
    num_keys = int(input())
    input_keys = list(map(int, input().split()))
    assert len(input_keys) == num_keys

    num_queries = int(input())
    input_queries = list(map(int, input().split()))
    assert len(input_queries) == num_queries

    for q in input_queries:
        print(binary_search(input_keys, q), end=' ')

```

```

from bisect import bisect_left

def binary_search(keys, query):
    i = bisect_left(keys, query)
    if i != len(keys) and keys[i] == q:
        return i
    return -1

if __name__ == '__main__':
    num_keys = int(input())
    input_keys = list(map(int, input().split()))
    assert len(input_keys) == num_keys

    num_queries = int(input())
    input_queries = list(map(int, input().split()))
    assert len(input_queries) == num_queries

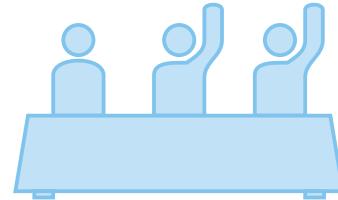
```

```
for q in input_queries:  
    print(binary_search(input_keys, q), end=' ')
```

### 6.2.3 Majority Element

#### Majority Element Problem

Check whether a given sequence of numbers contains an element that appears more than half of the times.



**Input:** A sequence of  $n$  integers.

**Output:** 1, if there is an element that is repeated more than  $n/2$  times, and 0 otherwise.

**Input format.** The first line contains an integer  $n$ , the next one contains a sequence of  $n$  non-negative integers.  $a_0, \dots, a_{n-1}$ .

**Output format.** Output 1 if the sequence contains an element that appears more than  $n/2$  times, and 0 otherwise.

**Constraints.**  $1 \leq n \leq 10^5$ ;  $0 \leq a_i \leq 10^9$  for all  $0 \leq i < n$ .

#### Sample 1.

Input:

```
5
2 3 9 2 2
```

Output:

```
1
```

2 is the majority element.

#### Sample 2.

Input:

```
4
1 2 3 1
```

Output:

```
0
```

This sequence does not have a majority element (note that the element 1 is not a majority element).

## Solution

Here is a naive algorithm for solving the Majority Element Problem with quadratic running time:

```
MAJORITYELEMENT( $A[1..n]$ ):  
for  $i$  from 1 to  $n$ :  
     $currentElement \leftarrow A[i]$   
     $count \leftarrow 0$   
    for  $j$  from 1 to  $n$ :  
        if  $A[j] = currentElement$ :  
             $count \leftarrow count + 1$   
    if  $count > n/2$ :  
        return 1  
return 0
```

In practice, one can scan the input sequence and save the number of occurrences of each element in an associative array. The running time of this solution depends on a particular implementation of an associative array. If it is implemented as a balanced search tree, every lookup in the array will cost  $O(\log n)$  and the overall running time will be  $O(n \log n)$ . For hash tables, the lookups are efficient in practice though may vary depending on the input data.

The divide-and-conquer strategy results in a simple algorithm with running time  $O(n \log n)$ . A simple, but crucial idea: *if  $e$  is a majority element in a sequence, then  $e$  must be a majority element in at least one of its halves.* Note that the converse is not true: both halves of a sequence  $(2, 3, 3, 7, 5, 7)$  contain majority elements (3 and 7, respectively), but none of them is a majority element of the original sequence. This leads to the following algorithm: find a majority element recursively in both halves and for each of them check its number of occurrences in the original sequence. For the last step, we need two linear scans that can be performed in time  $O(n)$ . Hence, the running time  $T(n)$  satisfies  $T(n) \leq 2T(n/2) + O(n)$  and thus  $T(n) = O(n \log n)$ .

**Exercise Break.** Can you design an even faster  $O(n)$  algorithm? It is based on the following idea. Partition the input elements into pairs. For each pair, if the two elements are different, discard both of them; otherwise, discard one of them.

## Code

The following code implements the divide-and-conquer algorithm discussed above. Note that we are using semiopen intervals for recursive calls (see [LR2](#)).

```
def majority_element(elements, left, right):
    if left == right:
        return -1
    if left + 1 == right:
        return elements[left]
    middle = (left + right) // 2
    x = majority_element(elements, left, middle)
    if 2 * elements[left:right].count(x) > right - left:
        return x
    x = majority_element(elements, middle, right)
    if 2 * elements[left:right].count(x) > right - left:
        return x
    return -1

if __name__ == '__main__':
    input_n = int(input())
    input_elements = list(map(int, input().split()))
    assert len(input_elements) == input_n
    if majority_element(input_elements, 0, input_n) != -1:
        print(1)
    else:
        print(0)
```

A solution that uses an associative array (or a dictionary).

```
from collections import defaultdict

def majority_element(elements):
    table = defaultdict(int)
    for e in elements:
```

```

        table[e] += 1
        if table[e] > len(elements) / 2:
            return 1

    return 0

if __name__ == '__main__':
    input_n = int(input())
    input_elements = list(map(int, input().split()))
    assert len(input_elements) == input_n
    print(majority_element(input_elements))

```

Finally, a solution where counting the number of occurrences of all elements is delegated to a built-in Python class Counter. We then just take the most frequent element and compare its number of occurrences with  $n/2$ .

```

from collections import Counter


def majority_element(elements):
    counter = Counter(elements)
    element, count = counter.most_common()[0]
    if count > len(elements) / 2:
        return 1
    else:
        return 0


if __name__ == '__main__':
    input_n = int(input())
    input_elements = list(map(int, input().split()))
    assert len(input_elements) == input_n
    print(majority_element(input_elements))

```

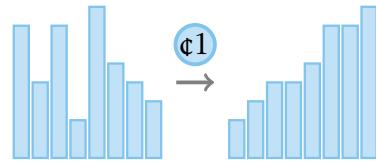
## 6.2.4 Speeding-up RANDOMIZEDQUICKSORT

### Speeding-up RANDOMIZEDQUICKSORT Problem

Sort a given sequence of numbers (that may contain duplicates) using a modification of RANDOMIZEDQUICKSORT that works in  $O(n \log n)$  expected time.

**Input:** An integer array with  $n$  elements that may contain duplicates.

**Output:** Sorted array (generated using a modification of RANDOMIZEDQUICKSORT) that works in  $O(n \log n)$  expected time.



Pavel, I've decided to repeat the pseudocode of quicksort below

In Section 2.7, we considered the RANDOMIZEDQUICKSORT 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 sorted array  $c_{sorted}$   
return  $c_{sorted}$ 
```

This pseudocode assumes that all elements of an array are different. The expected running time of the algorithm is  $O(n \log n)$ .

It is easy to modify the algorithm for the case when this array contains duplicates. To do this, let  $c_{small}$  contain all elements that are *at most*  $m$  (rather than smaller than  $m$ ). However, this modification becomes slow (even with respect to the expected running time!). For example, when all elements of  $c$  are the same, the partition procedure splits  $c$  into two parts:

$c_{small}$  has size  $n - 1$ , whereas  $c_{large}$  is empty. Since RANDOMIZEDQUICKSORT spends  $a \cdot n$  time to perform this partition, its overall running time is:

$$a \cdot n + a \cdot (n - 1) + a \cdot (n - 2) + \dots = a \cdot \frac{n \cdot (n + 1)}{2},$$

that is,  $O(n^2)$  instead of  $O(n \log n)$ .

Your goal is to modify the RANDOMIZEDQUICKSORT algorithm described above so that it works in  $O(n \log n)$  expected running time even on sequences containing many repeated elements.

**Input format.** The first line of the input contains an integer  $n$ . The next line contains a sequence of  $n$  integers  $a_0, a_1, \dots, a_{n-1}$ .

**Output format.** Output this sequence sorted in non-decreasing order.

**Constraints.**  $1 \leq n \leq 10^5$ ;  $1 \leq a_i \leq 10^9$  for all  $0 \leq i < n$ .

**Sample.**

Input:

```
5
2 3 9 2 2
```

Output:

```
2 2 2 3 9
```

**Programming tips:** do not use built-in sorting algorithms ([LC6](#)).

## Solution

To speed-up RANDOMIZEDQUICKSORT, we will split the input array into three subarrays: elements that are smaller, equal, and greater than the pivot. In the code below, we do this naively: we scan the array three times to collect the required elements.

**Exercise Break.** Show how to partition the array into three parts (smaller, equal, and greater than the pivot) *in-place*, i.e., without allocating additional memory.

## Code

```
from random import randint

def quick_sort(lst):
    if len(lst) <= 1:
        return lst

    pivot = lst[randint(0, len(lst) - 1)]

    smaller = [x for x in lst if x < pivot]
    equal = [x for x in lst if x == pivot]
    larger = [x for x in lst if x > pivot]

    smaller = quick_sort(smaller)
    larger = quick_sort(larger)

    return smaller + equal + larger

if __name__ == '__main__':
    input_n = int(input())
    elements = list(map(int, input().split()))
    assert len(elements) == input_n
    print(*quick_sort(elements))
```

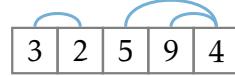
## 6.2.5 Number of Inversions

### Number of Inversions Problem

Compute the number of inversions in a sequence of integers.

**Input:** A sequence of  $n$  integers  $a_1, \dots, a_n$ .

**Output:** The number of inversions in the sequence, i.e., the number of indices  $i < j$  such that  $a_i > a_j$ .



The number of inversions in a sequence measures how close the sequence is to being sorted. For example, a sequence sorted in the non-descending order contains no inversions, while a sequence sorted in the descending order contains  $n(n - 1)/2$  inversions (every two elements form an inversion).

A naive algorithm for the Number of Inversions Problem goes through all possible pairs  $(i, j)$  and has running time  $O(n^2)$ . To solve this problem in time  $O(n \log n)$  using the divide-and-conquer technique split the input array into two halves and make a recursive call on both halves. What remains to be done is computing the number of inversions formed by two elements from different halves. If we do this naively, this will bring us back to  $O(n^2)$  running time, since the total number of such pairs is  $\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4} = O(n^2)$ . It turns out that one can compute the number of inversions formed by two elements from different halves in time  $O(n)$ , if both halves are already sorted. This suggest that instead of solving the original problem we solve a more general problem: compute the number of inversions in the given array and sort it at the same time.

**Exercise Break.** Modify the MERGESORT algorithm for solving this problem.

**Input format.** The first line contains an integer  $n$ , the next one contains a sequence of integers  $a_1, \dots, a_n$ .

**Output format.** The number of inversions in the sequence.

**Constraints.**  $1 \leq n \leq 30\,000$ ,  $1 \leq a_i \leq 10^9$  for all  $1 \leq i \leq n$ .

**Sample.**

Input:

```
5
2 3 9 2 9
```

Output:

```
2
```

The two inversions here are  $(2, 4)$  ( $a_2 = 3 > 2 = a_4$ ) and  $(3, 4)$  ( $a_3 = 9 > 2 = a_4$ ).

**Programming tips:** for recursive implementations, use semiopen intervals ([LR2](#)).

**Solution**

Let us try the most frequently used type of the divide-and-conquer strategy: split the input sequence into two halves, *LeftHalf* and *RightHalf*, and make a recursive call for each of them. This allows us to compute all inversions that lie in the same half. But it does not reveal the numbers of *split inversions*, i.e., the number of pairs  $(a_i, a_j)$  such that  $a_i$  lies in the left half,  $a_j$  lies in the right half, and  $a_i > a_j$ .

**Stop and Think.** Consider an element  $x$  in *LeftHalf*. What is the number of split inversions that  $x$  belongs to?

Given an array *List* and an integer  $x$ , let  $\text{List}_x$  be the number of elements in *List* that are smaller than  $x$ . Since the answer to the Stop and Think question above is  $\text{RightHalf}_x$ , our goal is to rapidly compute  $\text{List}_x$ .

It is equal to the number of elements in *RightHalf* that are smaller than  $x$ . This way, we face the following problem: given a sequence of integers *List* and an integer  $x$ , find the number of elements in *List* that are smaller than  $x$ . One can do it in  $O(|\text{List}|)$  time in the case of unordered array (since each element of the array has to be checked) and in  $O(\log |\text{List}|)$  time in the case of an ordered array by using the binary search.

**Exercise Break.** Show how to implement a method `COUNTSMALLER(List, x)` for counting the number of elements of *List* that are smaller than  $x$  in time  $O(\log_2 |\text{List}|)$ .

This way, we arrive at the following divide-and-conquer algorithm.

```
COUNTINVERSIONS(List):
if |List| ≤ 1:
    return 0
inversions ← 0
LeftHalf ← left half of List
RightHalf ← right half of List
inversions ← inversions + COUNTINVERSIONS(LeftHalf)
inversions ← inversions + COUNTINVERSIONS(RightHalf)
sort RightHalf
for x in LeftHalf:
    inversions ← inversions + COUNTSMALLER(RightHalf, x)
return inversions
```

The running time  $T(n)$  (where  $n$  is the length of *List*) satisfies a recurrence relation

$$T(n) \leq 2T(n/2) + O(n \log n).$$

The  $O(n \log n)$  term includes two things: sorting *RightHalf* and answering  $n/2$  COUNTSMALLER queries. This recurrence cannot be plugged into the Master Theorem directly as the term  $O(n \log n)$  is not of the form  $O(n^d)$  for a constant  $d$ . Still, one can analyze this recurrence in the same fashion: the recursion tree has  $\log_2 n$  levels, the total size of all problems on every level is equal to  $n$ , and the total time spent on every level is  $O(n \log n)$ . Thus, the total running time is  $O(n \log^2 n)$ . Instead of formally proving it, we will improve the above algorithm so that it works in time  $O(n \log n)$ .

One can find all split inversions quickly, if together with counting inversions one sorts an input sequence. That is, assume that an algorithm COUNTINVERSIONSANDSORT(*List*) returns the numbers of inversions in *List* and sorts *List*. After two recursive calls, both halves of *List* are sorted. At this point, we need to do two things: sort the whole sequence *List* and to compute the number of split inversions. We already know how to achieve the first goal: the Merge procedure is responsible for this (recall Section 2.6). It proceeds as follows. Let  $l$  and  $r$  be the first elements of (sorted) sequences *LeftHalf* and *RightHalf*. It chooses the smallest out of them and moves it to the growing sorted list.

**Stop and Think.** Can you find the number of split inversions that the moved element forms?

Consider two cases.

- $l \leq r$ . In this case,  $l$  is not greater than every element of  $RightHalf$  and hence forms no split inversions.
- $l > r$ . In this case,  $r$  is smaller than every element of  $LeftHalf$  and hence forms a split inversion with every such element.

This leads us to the following extension of the MERGE method.

```
MERGE(LeftHalf, RightHalf):
SortedList  $\leftarrow$  empty list
inversions  $\leftarrow$  0
while both LeftHalf and RightHalf are non-empty:
    l  $\leftarrow$  the first element of LeftHalf
    r  $\leftarrow$  the first element of RightHalf
    if l  $\leq$  r:
        move l from LeftHalf to SortedList
    else:
        move r from RightHalf to SortedList
        inversions  $\leftarrow$  inversions + |LeftHalf|
append LeftHalf and RightHalf to SortedList
return SortedList, inversions
```

And this is the final algorithm.

```
SORTANDCOUNTINVERSIONS(List):
if |List|  $\leq$  1:
    return 0
LeftHalf  $\leftarrow$  left half of List
RightHalf  $\leftarrow$  right half of List
leftInv  $\leftarrow$  SORTANDCOUNTINVERSIONS(LeftHalf)
rightInv  $\leftarrow$  SORTANDCOUNTINVERSIONS(RightHalf)
List, splitInv  $\leftarrow$  MERGE(LeftHalf, RightHalf)
return leftInv + rightInv + splitInv
```

The running time  $T(n)$  of this algorithm satisfies a recurrence  $T(n) = 2T(n/2) + O(n)$  and hence  $T(n) = O(n \log n)$ .

## Code

```
from bisect import bisect_left

def compute_inversions(lst):
    if len(lst) <= 1:
        return 0

    left_lst = lst[:len(lst) // 2]
    right_lst = lst[len(lst) // 2:]
    left_inv = compute_inversions(left_lst)
    right_inv = compute_inversions(right_lst)

    split_inv = 0
    right_lst = sorted(right_lst)
    for a in left_lst:
        split_inv += bisect_left(right_lst, a)

    return left_inv + right_inv + split_inv

if __name__ == '__main__':
    input_n = int(input())
    elements = list(map(int, input().split()))
    assert len(elements) == input_n
    print(compute_inversions(elements))
```

```
def merge(left, right):
    sorted_lst, inversions = [], 0

    while len(left) > 0 and len(right) > 0:
        if left[0] <= right[0]:
            sorted_lst.append(left.pop(0))
        else:
            sorted_lst.append(right.pop(0))
            inversions += len(left)
```

```
        inversions += len(left)

    sorted_lst.extend(left)
    sorted_lst.extend(right)

    return sorted_lst, inversions

def sort_and_compute_inversions(lst):
    if len(lst) <= 1:
        return lst, 0

    left_lst = lst[:len(lst) // 2]
    right_lst = lst[len(lst) // 2:]
    left_lst, left_inv = sort_and_compute_inversions(left_lst)
    right_lst, right_inv = sort_and_compute_inversions(right_lst)

    lst, split_inv = merge(left_lst, right_lst)

    return lst, left_inv + right_inv + split_inv

if __name__ == '__main__':
    input_n = int(input())
    elements = list(map(int, input().split()))
    assert len(elements) == input_n
    _, inversions = sort_and_compute_inversions(elements)
    print(inversions)
```

## 6.2.6 Organizing a Lottery

---

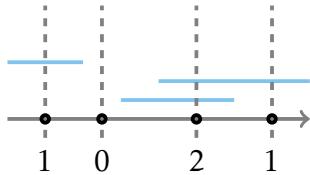
### Points and Segments Problem

*Given a set of points and a set of segments on a line, compute, for each point, the number of segments it is contained in.*

**Input:** A list of  $n$  segments and a list of  $m$  points.

**Output:** The number of segments containing each point.

---



You are organizing an online lottery. To participate, a person bets on a single integer. You then draw several segments of consecutive integers at random. A participant's payoff is proportional to the number of segments that contain the participant's number. You need an efficient algorithm for computing the payoffs for all participants. A simple scan of the list of all ranges for each participant is too slow since your lottery is very popular: you have thousands of participants and thousands of ranges.

**Input format.** The first line contains two non-negative integers  $n$  and  $m$  defining the number of segments and the number of points on a line, respectively. The next  $n$  lines contain two integers  $l_i, r_i$  defining the  $i$ -th segment  $[l_i, r_i]$ . The next line contains  $m$  integers defining points  $p_1, \dots, p_m$ .

**Output format.**  $m$  non-negative integers  $k_1, \dots, k_p$  where  $k_i$  is the number of segments that contain  $p_i$ .

**Constraints.**  $1 \leq n, m \leq 50\,000$ ;  $-10^8 \leq l_i \leq r_i \leq 10^8$  for all  $1 \leq i \leq n$ ;  $-10^8 \leq p_j \leq 10^8$  for all  $1 \leq j \leq m$ .

### Sample 1.

Input:

```
2 3
0 5
7 10
1 6 11
```

Output:

```
1 0 0
```

We have two segments and three points. The first point lies only in the first segment while the remaining two points are outside of all segments.

### Sample 2.

Input:

```
1 3
-10 10
-100 100 0
```

Output:

```
0 0 1
```

### Sample 3.

Input:

```
3 2
0 5
-3 2
7 10
1 6
```

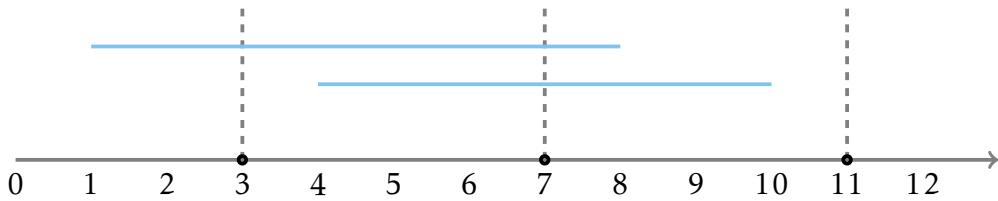
Output:

```
2 0
```

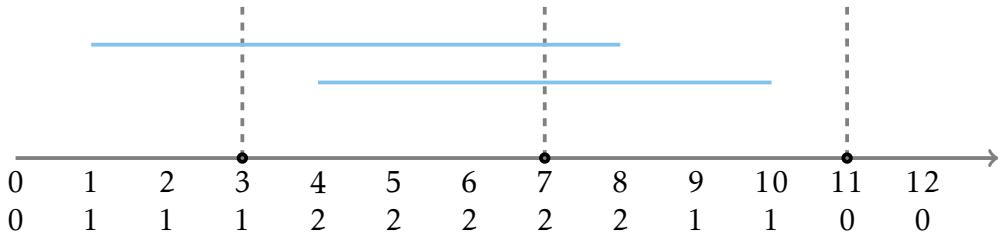
**Programming tips:** use built-in binary search and sorting algorithms if needed ([LC6](#)).

### Solution 1: Sorting All Points

Consider an example with  $n = 2$  segments  $[l_1, r_1] = [4, 10]$ ,  $[l_2, r_2] = [1, 8]$  and  $m = 3$  points  $p_1 = 11$ ,  $p_2 = 7$ ,  $p_3 = 3$ :

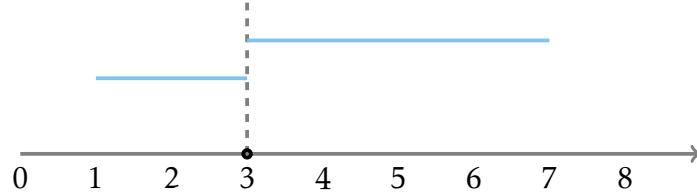


Let's now initialize  $numberOfSegments$  to 0 and scan the line above from the left to the right. We will increase (decrease)  $numberOfSegments$  by 1 each time we encounter a left (right) end-point of an interval. Initially,  $numberOfSegments$  is set to 0. We arrive to  $l_2 = 1$  and increment  $numberOfSegments$ . We then move on to  $p_3$ . This is a point from the dataset (rather than an end-point of a segment), so we don't change the value of  $numberOfSegments$ , but we now know that  $p_3 = 3$  is covered by a single segment. We then proceed to  $l_1$  and increment  $numberOfSegments$  again. We then meet another point  $p_2$ . As with the previous point, we know that it is covered by  $numberOfSegments = 2$  segments. We move on to  $r_2$  and decrement  $numberOfSegments$ . We decrement it once again when we reach  $r_1$ . Finally, we end our journey at  $p_2$  that is covered by  $numberOfSegments = 0$  segments. The value of  $numberOfSegments$  for each point is shown below.



Intuitively, we are moving along a line maintaining the invariant that  $numberOfSegments$  is equal to the number of segments containing a given point. For a given point, the segments that contain it start to the left of the point and end to the right of the point. And this is exactly what we count. Note that each segment that starts and ends to the left of the point first contributes +1 to  $numberOfSegments$ , but then cancels this by contributing -1.

The only remaining issue is how to handle coinciding points and segment ends. For example, what if  $[l_1, r_1] = [1, 3]$ ,  $p_1 = 3$ , and  $[l_2, r_2] = [3, 7]$ ?



That is, there are three different objects at the point with coordinate 3: the end  $r_1 = 3$  of the first segment, the point  $p_1 = 3$ , and the beginning of the second segment  $l_2 = 3$ . It is important to process these three events in the following order:

1. We first process  $l_2$  and set  $\text{numberOfSegments}$  to 2.
2. We then report that  $p_1$  is covered by  $\text{numberOfSegments} = 2$  segments.
3. We then process  $r_1$  and decrement  $\text{numberOfSegments}$ .

In other words, for a given point we need to first process all segments that start at this point, then report the number of segments covering it, and then process all segments that end at this point. To make sure that we process all the events in this order, we use 'l', 'p', and 'r' as indicators for left-ends, points, and right-ends in the code below. The needed order is then guaranteed by the lexicographical ordering of these indicators since  $l < p < r$ .

The running time is  $O((n+m)\log(n+m))$  since, as we explain later in this book, an array with  $k$  elements can be sorted in  $O(k \log k)$  time.

## Solution 2: Binary Search

Let  $\text{before}(p)$  be the number of segments that end before a point  $p$ ,  $\text{after}(p)$  be the number of segments that start after  $p$ , and  $\text{cover}(p)$  be the number of segments covering  $p$ .

**Exercise Break.** Prove that for each point  $p$ ,  $\text{before}(p) + \text{after}(p) + \text{cover}(p)$  is equal to the total number of segments.

Hence, to count the number of segments that do not cover the given point  $p$ , it is sufficient to count the number of right-ends of segments that are smaller than  $p$  and the number of left-ends of segments that are

greater than  $p$ . If all left-ends and right-ends are sorted, one can use the binary search algorithm to perform such a check in  $O(\log n)$  time. The corresponding solution has running time  $O(m \log m + n \log m)$ .

## Code

```
from sys import stdin
from collections import namedtuple

Event = namedtuple('Event', ['coordinate', 'type', 'index'])

def points_cover(starts, ends, points):
    count = [None] * len(points)

    events = []
    for i in range(len(starts)):
        events.append(Event(starts[i], 'l', i))
        events.append(Event(ends[i], 'r', i))
    for i in range(len(points)):
        events.append(Event(points[i], 'p', i))

    events = sorted(events)
    number_of_segments = 0
    for e in events:
        if e.type == 'l':
            number_of_segments += 1
        elif e.type == 'r':
            number_of_segments -= 1
        elif e.type == 'p':
            count[e.index] = number_of_segments
        else:
            assert False

    return count

if __name__ == '__main__':
```

```

data = list(map(int, stdin.read().split()))
n, m = data[0], data[1]
starts, ends = data[2:2 * n + 2:2], data[3:2 * n + 2:2]
points = data[2 * n + 2:]

count = points_cover(starts, ends, points)
print(*count)

```

```

from sys import stdin
from bisect import bisect_left, bisect_right


def points_cover(starts, ends, points):
    starts, ends = sorted(starts), sorted(ends)

    count = [len(starts)] * len(points)
    for index, point in enumerate(points):
        count[index] -= bisect_left(ends, point)
        count[index] -= len(starts) - bisect_right(starts, point)

    return count

if __name__ == '__main__':
    data = list(map(int, stdin.read().split()))
    n, m = data[0], data[1]
    starts, ends = data[2:2 * n + 2:2], data[3:2 * n + 2:2]
    points = data[2 * n + 2:]

    count = points_cover(starts, ends, points)
    print(*count)

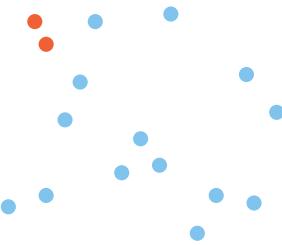
```

### 6.2.7 Closest Points

---

#### Closest Points Problem

*Find the closest pair of points in a set of points on a plane.*



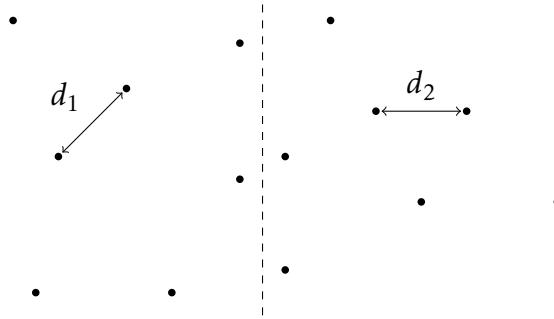
**Input:** A list of  $n$  points on a plane.

**Output:** The minimum distance between a pair of these points.

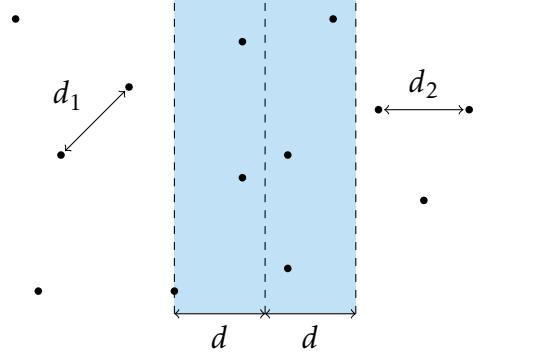
---

This computational geometry problem has many applications in computer graphics and vision. A naive algorithm with quadratic running time iterates through all pairs of points to find the closest pair. Your goal is to design an  $O(n \log n)$  time divide and conquer algorithm.

To solve this problem in time  $O(n \log n)$ , let's first split the given  $n$  points by an appropriately chosen vertical line into two halves  $S_1$  and  $S_2$  of size  $\frac{n}{2}$  (assume for simplicity that all  $x$ -coordinates of the input points are different). By making two recursive calls for the sets  $S_1$  and  $S_2$ , we find the minimum distances  $d_1$  and  $d_2$  in these subsets. Let  $d = \min\{d_1, d_2\}$ .



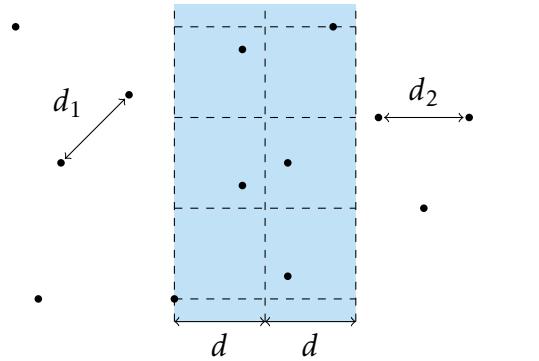
It remains to check whether there exist points  $p_1 \in S_1$  and  $p_2 \in S_2$  such that the distance between them is smaller than  $d$ . We cannot afford to check all possible such pairs since there are  $\frac{n}{2} \cdot \frac{n}{2} = \Theta(n^2)$  of them. To check this faster, we first discard all points from  $S_1$  and  $S_2$  whose  $x$ -distance to the middle line is greater than  $d$ . That is, we focus on the following strip:



**Stop and Think.** Why can we narrow the search to this strip?

Now, let's sort the points of the strip by their  $y$ -coordinates and denote the resulting sorted list by  $P = [p_1, \dots, p_k]$ . It turns out that if  $|i - j| > 7$ , then the distance between points  $p_i$  and  $p_j$  is greater than  $d$  for sure. This follows from the Exercise Break below.

**Exercise Break.** Partition the strip into  $d \times d$  squares as shown below and show that each such square contains at most four input points.



This results in the following algorithm. We first sort the given  $n$  points by their  $x$ -coordinates and then split the resulting sorted list into two halves  $S_1$  and  $S_2$  of size  $\frac{n}{2}$ . By making a recursive call for each of the sets  $S_1$  and  $S_2$ , we find the minimum distances  $d_1$  and  $d_2$  in them. Let  $d = \min\{d_1, d_2\}$ . However, we are not done yet as we also need to find the minimum distance between points from different sets (i.e., a point from  $S_1$  and a point from  $S_2$ ) and check whether it is smaller than  $d$ . To perform

such a check, we filter the initial point set and keep only those points whose  $x$ -distance to the middle line does not exceed  $d$ . Afterward, we sort the set of points in the resulting strip by their  $y$ -coordinates and scan the resulting list of points. For each point, we compute its distance to the seven subsequent points in this list and compute  $d'$ , the minimum distance that we encountered during this scan. Afterward, we return  $\min\{d, d'\}$ .

The running time of the algorithm satisfies the recurrence relation

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n \log n).$$

The  $O(n \log n)$  term comes from sorting the points in the strip by their  $y$ -coordinates at every iteration.

**Exercise Break.** Prove that  $T(n) = O(n \log^2 n)$  by analyzing the recursion tree of the algorithm.

**Exercise Break.** Show how to bring the running time down to  $O(n \log n)$  by avoiding sorting at each recursive call.

**Input format.** The first line contains the number of points  $n$ . Each of the following  $n$  lines defines a point  $(x_i, y_i)$ .

**Output format.** The minimum distance. Recall that the distance between points  $(x_1, y_1)$  and  $(x_2, y_2)$  is equal to  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ . Thus, while the Input contains only integers, the Output is not necessarily integer and you have to pay attention to precision when you report it. The absolute value of the difference between the answer of your program and the optimal value should be at most  $10^{-3}$ . To ensure this, output your answer with at least four digits after the decimal point (otherwise even correctly computed answer may fail to pass our grader because of the rounding errors).

**Constraints.**  $2 \leq n \leq 10^5$ ;  $-10^9 \leq x_i, y_i \leq 10^9$  are integers.

**Sample 1.**

Input:

```
2  
0 0  
3 4
```

Output:

```
5.0
```

There are only two points at distance 5.

**Sample 2.**

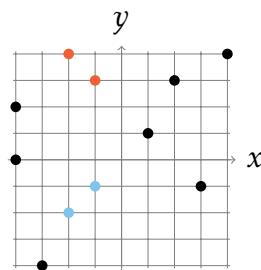
Input:

```
11  
4 4  
-2 -2  
-3 -4  
-1 3  
2 3  
-4 0  
1 1  
-1 -1  
3 -1  
-4 2  
-2 4
```

Output:

```
1.414213
```

The smallest distance is  $\sqrt{2}$ . There are two pairs of points at this distance shown in blue and red below:  $(-1, -1)$  and  $(-2, -2)$ ;  $(-2, 4)$  and  $(-1, 3)$ .



**Programming tips:** avoid using floating point numbers whenever possible ([LI3](#)), use built-in sorting algorithms if needed ([LC6](#)).

## Code

```
from collections import namedtuple
from itertools import combinations
from math import sqrt

Point = namedtuple('Point', 'x y')

def distance(first_point, second_point):
    return (first_point.x - second_point.x) ** 2 + \
           (first_point.y - second_point.y) ** 2

def get_min_distance(points):
    if len(points) <= 4:
        min_dist = float("inf")

        for p, q in combinations(points, 2):
            min_dist = min(min_dist, distance(p, q))

        return min_dist
    else:
        sorted_points = sorted(points, key=lambda p: p.x)
        mid_index = len(sorted_points) // 2
        mid_x = sorted_points[mid_index].x

        left_dist = get_min_distance(sorted_points[:mid_index])
        right_dist = get_min_distance(sorted_points[mid_index:])
        min_dist = min(left_dist, right_dist)

        strip = [p for p in points
                 if abs(p.x - mid_x) ** 2 < min_dist]
```

```
strip = sorted(strip, key=lambda p: p.y)
for i in range(len(strip)):
    j = i + 1
    while j < len(strip) and \
          distance(strip[i], strip[j]) < min_dist:
        min_dist = distance(strip[i], strip[j])
        j += 1

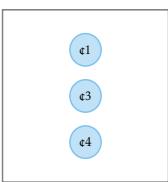
return min_dist

if __name__ == '__main__':
    n = int(input())
    points = []
    for _ in range(n):
        x, y = map(int, input().split())
        input_point = Point(x, y)
        points.append(input_point)

print('{0:.9f}'.format(sqrt(get_min_distance(points))))
```

# Chapter 7: Dynamic Programming

In this chapter, you will implement various dynamic programming algorithms and will see how they solve problems that evaded all attempts to solve them using greedy or divide-and-conquer strategies. There are countless applications of dynamic programming in practice ranging from searching for similar Internet pages to gene prediction in DNA sequences. You will learn how the same idea helps to automatically make spelling corrections and to find the differences between two versions of the same text.

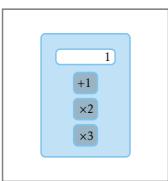


## Money Change Again

*Compute the minimum number of coins needed to change the given value into coins with denominations 1, 3, and 4.*

**Input.** An integer *money*.

**Output.** The minimum number of coins with denominations 1, 3, and 4 that changes *money*.

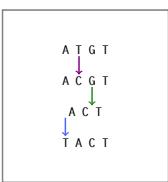


## Primitive Calculator

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

**Input.** An integer *n*.

**Output.** The minimum number of operations “+1”, “ $\times 2$ ”, and “ $\times 3$ ” needed to get *n* from 1.

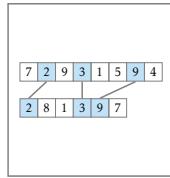


## Edit Distance

*Compute the edit distance between two strings.*

**Input.** Two strings.

**Output.** The minimum number of single-symbol insertions, deletions, and substitutions to transform one string into the other one.

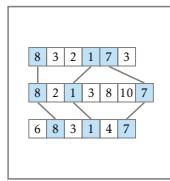


### Longest Common Subsequence of Two Sequences

*Compute the maximum length of a common subsequence of two sequences.*

Input. Two sequences.

Output. The maximum length of a common subsequence.

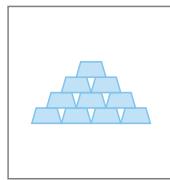


### Longest Common Subsequence of Three Sequences

*Compute the maximum length of a common subsequence of three sequences.*

Input. Three sequences.

Output. The maximum length of a common subsequence.

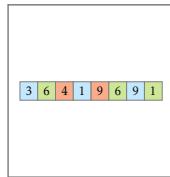


### Maximum Amount of Gold

*Given a set of gold bars of various weights and a backpack that can hold at most  $W$  pounds, place as much gold as possible into the backpack.*

Input. A set of  $n$  gold bars of integer weights  $w_1, \dots, w_n$  and a backpack that can hold at most  $W$  pounds.

Output. A subset of gold bars of maximum total weight not exceeding  $W$ .



### 3-Partition

*Partition a set of integers into three subsets with equal sums.*

Input. A sequence of integers  $v_1, v_2, \dots, v_n$ .

Output. Check whether it is possible to partition them into three subsets with equal sums, i.e., check whether there exist three disjoint sets  $S_1, S_2, S_3 \subseteq \{1, 2, \dots, n\}$  such that  $S_1 \cup S_2 \cup S_3 = \{1, 2, \dots, n\}$  and

$$\sum_{i \in S_1} v_i = \sum_{j \in S_2} v_j = \sum_{k \in S_3} v_k.$$

$$\begin{array}{l} ((8 - 5) \times 3) = 9 \\ \quad \uparrow \\ 8 - 5 \times 3 \\ \downarrow \\ (8 - (5 \times 3)) = -7 \end{array}$$

### Maximum Value of an Arithmetic Expression

*Parenthesize an arithmetic expression to maximize its value.*

**Input.** An arithmetic expression consisting of digits as well as plus, minus, and multiplication signs.

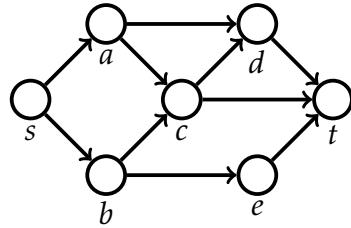
**Output.** Add parentheses to the expression in order to maximize its value.

## 7.1 The Main Idea

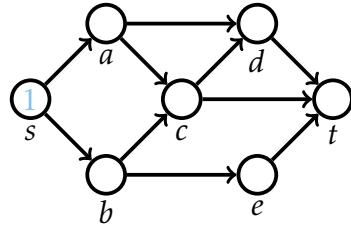
### 7.1.1 Number of Paths

To “invent” the key idea of the dynamic programming technique, try to solve the following puzzle.

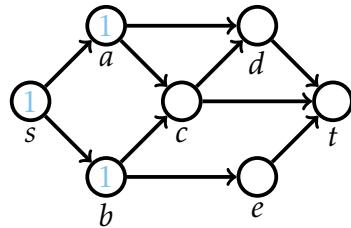
**Interactive Puzzle “Number of Paths”.** There are many ways of getting from  $s$  to  $t$  in the network below: for example,  $s \rightarrow b \rightarrow e \rightarrow t$  and  $s \rightarrow a \rightarrow c \rightarrow d \rightarrow t$ . What is the total number of paths? [Try it online \(level 1\)!](#)



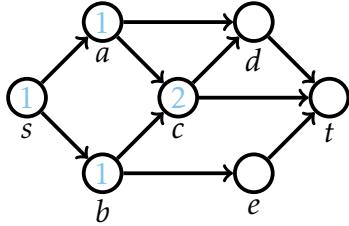
Since we start from  $s$ , there is a unique way to get to  $s$ . Let’s write this down:



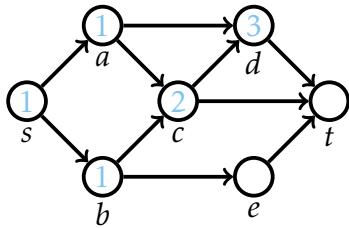
For  $a$  and  $b$ , there is also just a single path.



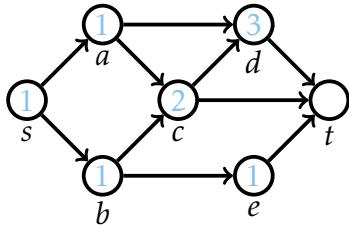
Since there is only one path to  $a$  and only one path to  $b$ , the number of paths to  $c$  is  $1 + 1 = 2$  ( $s \rightarrow a \rightarrow c$  and  $s \rightarrow b \rightarrow c$ ).



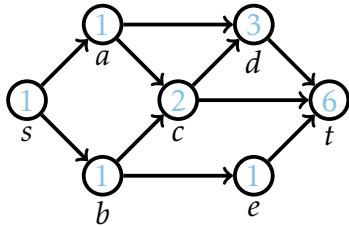
Similarly, to get to  $d$  one needs to get to either  $a$  or  $c$ . There is one path to get to  $a$  and two paths to get to  $c$ . Hence, the number of paths to get to  $d$  is  $1 + 2 = 3$  ( $s \rightarrow a \rightarrow d$ ,  $s \rightarrow a \rightarrow c \rightarrow d$ , and  $s \rightarrow b \rightarrow c \rightarrow d$ ).



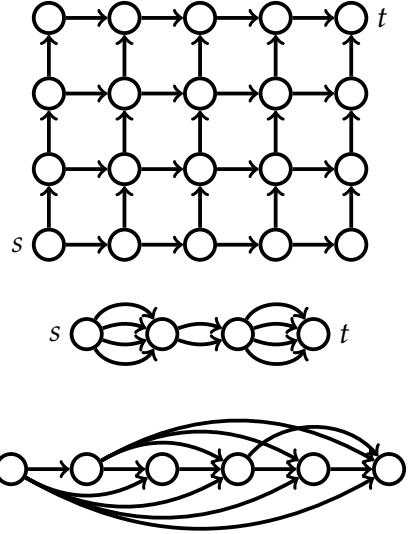
The number of paths ending in  $e$  is equal to 1 as  $e$  can be reached from  $b$  only.



Since there are two paths to  $c$ , three paths to  $d$ , and one path to  $e$ , there are  $2 + 3 + 1 = 6$  paths to  $t$ .



**Exercise Break.** Find the number of ways to get from  $s$  to  $t$  in the following three networks. [Try it online \(levels 2–4\)!](#)



**Exercise Break.** After finding the number of paths in the  $5 \times 4$  grid (shown above) by filling in the numbers in all nodes of this grid, can you propose a formula for the number of such paths? What is the number of such paths in an  $n \times m$  grid?

### 7.1.2 Dynamic Programming

Let's review our solution of the Number of Paths puzzle to state the main ideas of dynamic programming. For a node  $v$ , let  $\text{paths}(v)$  be the number of paths from  $s$  to a node  $v$ . Clearly,  $\text{paths}(s) = 1$ . This is called a *base case*. For all other nodes, the corresponding value can be found using a *recurrence relation*:

$$\text{paths}(v) = \sum_{\text{each predecessor } w \text{ of } v} \text{paths}(w),$$

where a predecessor of  $v$  is a node that has an edge connecting it with  $v$ .

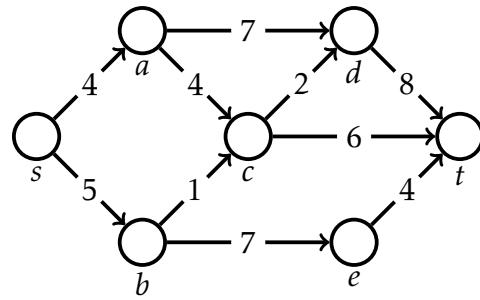
Many dynamic programming algorithms follow the same pattern:

- Instead of solving the original problem, the algorithm solves a bunch of subproblems of the same type.
- The algorithm computes a solution to every subproblem through a recurrence relation involving solutions to smaller subproblems.

- The algorithm stores solutions to subproblems to avoid recomputing them again.

### 7.1.3 Shortest Path in Directed Acyclic Graph

Now, consider a *weighted graph* where each edge  $e$  has length denoted  $\text{length}(e)$ . The length of a path in the graph is defined as the sum of its edge-lengths.



For example, the length of a path  $s \rightarrow b \rightarrow e \rightarrow t$  is  $5 + 7 + 4 = 16$ . What is the minimum length of a path from  $s$  to  $t$ ?

Since each path from  $s$  to  $t$  passes through either  $c$ ,  $d$ , or  $e$  before entering into  $t$ ,

$$\text{length}(t) = \min\{\text{length}(c) + 6, \text{length}(d) + 8, \text{length}(e) + 4\},$$

where  $\text{length}(v)$  is the minimum length of a path from  $s$  to  $v$ . The distances to  $c$ ,  $d$ , and  $e$  can be found using similar recurrence relations:

$$\begin{aligned} \text{length}(c) &= \min\{\text{length}(a) + 4, \text{length}(b) + 1\}, \\ \text{length}(d) &= \min\{\text{length}(a) + 7, \text{length}(c) + 2\}, \\ \text{length}(e) &= \text{length}(b) + 7. \end{aligned}$$

The recurrence relations for  $a$  and  $b$  are the following:

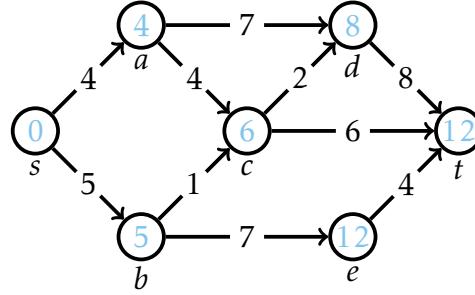
$$\begin{aligned} \text{length}(a) &= \text{length}(s) + 4, \\ \text{length}(b) &= \text{length}(s) + 5. \end{aligned}$$

Finally, the base case is  $\text{length}(s) = 0$ . Using this base case, one can find the distances to all the nodes in the network, including the target node  $t$ ,

through the recurrence relations given above. All of them can be compactly written as follows:

$$\text{length}(v) = \min_{\text{each predecessor } w \text{ of } v} \{\text{length}(w) + \text{length}(w, v)\}.$$

For our toy example, it is convenient to write the results down as we compute it right in the picture. The results looks like this.



**Stop and Think.** The minimum length of a path from  $s$  to  $t$  is 12. Do you see how to find a path of this length?

In dynamic programming algorithms, this is done by backtracking the choices that led to an optimum result. Specifically, let's highlight one of the three choices that leads to the value of  $\text{length}(t)$ :

$$\text{length}(t) = \min\{\text{length}(c) + 6, \text{length}(d) + 8, \text{length}(e) + 4\} = \min\{12, 16, 16\} = 12.$$

From this, we conclude that the last edge of an optimum path is  $c \rightarrow t$ . Similarly,

$$\text{length}(c) = \min\{\text{length}(a) + 4, \text{length}(b) + 1\} = \min\{8, 6\} = 6,$$

hence, we arrive to  $c$  from  $b$ . Thus, the path from  $s$  to  $t$  of length 12 is

$$s \rightarrow b \rightarrow c \rightarrow t.$$

A convenient property of the network above is that we were able to specify an order of its nodes ensuring the following property: every node goes after all its *predecessors*, that is, nodes that point to the current node (for example,  $c$ ,  $d$ , and  $e$  are predecessors of  $t$ ). Networks with this property are known as *directed acyclic graphs* or *DAGs*. We will see that many dynamic programming algorithms exploit DAGs, explicitly or implicitly.

## 7.2 Programming Challenges

### 7.2.1 Money Change Again

---

#### Money Change Again Problem

Compute the minimum number of coins needed to change the given value into coins with denominations 1, 3, and 4.



**Input:** An integer *money*.

**Output:** The minimum number of coins with denominations 1, 3, and 4 that changes *money*.

---

As we already know, a natural greedy strategy for the change problem does not work correctly for any set of denominations. For example, for denominations 1, 3, and 4, the greedy algorithm will change 6 cents using three coins ( $4 + 1 + 1$ ) while it can be changed using just two coins ( $3 + 3$ ). Your goal now is to apply dynamic programming for solving the Money Change Problem for denominations 1, 3, and 4.

**Input format.** Integer *money*.

**Output format.** The minimum number of coins with denominations 1, 3, and 4 that changes *money*.

**Constraints.**  $1 \leq \text{money} \leq 10^3$ .

**Sample.**

Input:

34

Output:

9

$34 = 3 + 3 + 4 + 4 + 4 + 4 + 4 + 4 + 4.$

#### Solution

An optimal way to change 26 requires seven coins. Let's consider an arbitrary subset of an optimal solution, e.g., four coins within a rectangle shown below sum up to 15.



**Stop and Think.** Can you change 15 cents with three coins?

The answer to this Stop and Think is “no” — if there was a way to change 15 with three coins, one would replace the highlighted four coins and get a change of 26 with six coins (rather than seven).

This toy example reveals an important property that many problems solved by the dynamic programming technique share:

*a solution to a problem contains solutions to all its smaller subproblems.*

This property allows one to find a solution for a problem by solving smaller subproblems first.

Let  $\text{change}(\text{money})$  be the minimum number of coins of denominations 1, 3, and 4 needed to change  $\text{money}$  and  $(c_1, \dots, c_k)$  be an optimal change for  $\text{money}$  so that

$$c_1 + \dots + c_k = \text{money}.$$

Then,

$$c_1 + \dots + c_{k-1} = \text{money} - c_k.$$

Hence,  $\text{change}(\text{money} - c_k) = k - 1$ . Thus, to solve the problem for  $\text{money}$ , it is enough to solve it for  $\text{money} - c_k$  and add one.

**Stop and Think.** Are we done?

The issue here is that we don't know the value of  $c_k$ . Still, we know that  $c_k$  is equal to either 1, or 3, or 4. Hence,  $\text{change}(\text{money})$  is equal to one of

- $\text{change}(\text{money} - 1) + 1$ ,
- $\text{change}(\text{money} - 3) + 1$ , and
- $\text{change}(\text{money} - 4) + 1$ .

Since we are looking for an optimum way to change,  $\text{money}$  is equal to the minimum of these three expressions. This leads us to the following recurrence relation:

$$\text{change}(\text{money}) = 1 + \min(\text{change}(\text{money} - c) : c \in \{1, 3, 4\}, c \leq \text{money}). \quad (7.1)$$

This relation expresses the values of the function *change* recursively through its own values on smaller arguments. For such a descending recursion, we need to specify a base case. In our case, it is  $money = 0$ :  $change(0) = 0$ .

The equation above is *the most important part of a dynamic programming algorithm*. In fact, it is straightforward to turn it into a recursive algorithm.

```
CHANGE(money):
if money = 0:
    return 0
else:
    result ← +∞
    for c = 1, 3, 4:
        if c ≤ money:
            result ← min(result, 1 + CHANGE(money - c))
    return result
```

There is a serious issue with this algorithm: it becomes prohibitively slow because it calls `CHANGE(money)` over and over again for the same value of *money*. A standard way to avoid this is known as *memoization* (we discussed it in the Fibonacci Number problem): when `CHANGE(money)` is computed, let's store it in a table so that we never need to compute it again.

```
table ← associative array

CHANGE(money):
if table[money] is not yet computed:
    if money = 0:
        table[money] ← 0
    else:
        result ← +∞
        for c = 1, 3, 4:
            if c ≤ money:
                result ← min(result, 1 + CHANGE(money - c))
        table[money] ← result
return table[money]
```

Such an algorithm is already good enough in practice, though still has

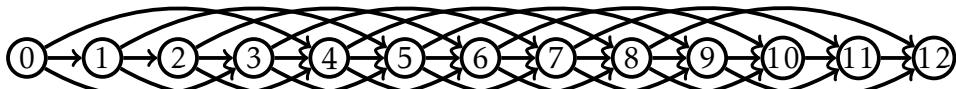
some inefficiencies: recursive calls and lookups in an associative array have an overhead. By noting that all the values that we need to compute are consecutive integers, we can implement a better approach that uses an array for storing solutions to all problems.

```
CHANGE(money):
table[0..money] ← [+∞, …, +∞]
table[0] ← 0

for  $m$  from 1 to  $money$ :
    for  $c = 1, 3, 4$ :
        if  $c \leq money$ :
            table[ $m$ ] ← min(table[ $m$ ], 1 + table[ $m - c$ ])
return table[ $money$ ]
```

The running time of this algorithm is  $O(money)$  as every iteration of the outer for loop takes constant time.

**Stop and Think.** Note that the algorithm described above implicitly finds a shortest graph in a DAG shown below (for  $money = 12$ ). Estimate the running time of the shortest path algorithm in a similar graph for an arbitrary value of  $money$ .



## Code

```
def change(money):
    table = [float('inf')] * (money + 1)
    table[0] = 0

    for m in range(1, money + 1):
        for coin in (1, 3, 4):
            if coin <= m:
                table[m] = min(table[m], table[m - coin] + 1)
```

```
return table[money]

if __name__ == '__main__':
    amount = int(input())
    print(change(amount))
```

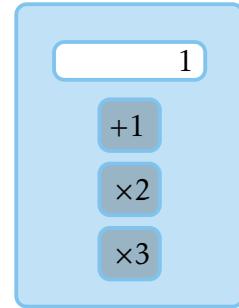
## 7.2.2 Primitive Calculator

### Primitive Calculator Problem

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

**Input:** An integer  $n$ .

**Output:** The minimum number of operations “+1”, “×2”, and “×3” needed to get  $n$  from 1.



You are given a calculator that only performs the following three operations with an integer  $x$ : add 1 to  $x$ , multiply  $x$  by 2, or multiply  $x$  by 3. Given a positive integer  $n$ , your goal is to find the minimum number of operations needed to obtain  $n$  starting from the number 1. Before solving the programming challenge below, test your intuition with our [Primitive Calculator](#) puzzle.

Let’s try a greedy strategy for solving this problem: if the current number is at most  $n/3$ , multiply it by 3; if it is larger than  $n/3$ , but at most  $n/2$ , multiply it by 2; otherwise add 1 to it. This results in the following pseudocode.

```
GREEDYCALCULATOR( $n$ ):  
    numOperations  $\leftarrow 0$   
    currentNumber  $\leftarrow 1$   
    while currentNumber  $< n$ :  
        if currentNumber  $\leq n/3$ :  
            currentNumber  $\leftarrow 3 \times$  currentNumber  
        else if currentNumber  $\leq n/2$ :  
            currentNumber  $\leftarrow 2 \times$  currentNumber  
        else:  
            currentNumber  $\leftarrow 1 +$  currentNumber  
        numOperations  $\leftarrow$  numOperations + 1  
    return numOperations
```

**Stop and Think.** Can you find a number  $n$  such that

**GREEDYCALCULATOR( $n$ )**

produces an incorrect result?

**Input format.** An integer  $n$ .

**Output format.** In the first line, output the minimum number  $k$  of operations needed to get  $n$  from 1. In the second line, output a sequence of intermediate numbers. That is, the second line should contain positive integers  $a_0, a_1, \dots, a_k$  such that  $a_0 = 1$ ,  $a_k = n$  and for all  $1 \leq i \leq k$ ,  $a_i$  is equal to either  $a_{i-1} + 1$ ,  $2a_{i-1}$ , or  $3a_{i-1}$ . If there are many such sequences, output any one of them.

**Constraints.**  $1 \leq n \leq 10^6$ .

**Sample 1.**

Input:

1

Output:

0

1

**Sample 2.**

Input:

96234

Output:

14

1 3 9 10 11 22 66 198 594 1782 5346 16038 16039 32078 96234

Another valid output in this case is “1 3 9 10 11 33 99 297 891 2673 8019 16038 16039 48117 96234”.

**Solution**

Let  $calculator(n)$  be the minimum number of operations needed to get  $n$  from 1. Since the last operation in an optimum sequence of operations is

“+1”, “×2”, or “×3”, we get the following recurrence relation, for  $n \geq 1$ :

$$\text{calculator}(n) = 1 + \min \begin{cases} \text{calculator}(n-1), \\ \text{calculator}(n/2), & \text{if } n \text{ is divisible by 2,} \\ \text{calculator}(n/3), & \text{if } n \text{ is divisible by 3.} \end{cases}$$

This recurrence relation, together with the base case  $\text{calculator}(1) = 1$ , can be mechanically converted into a recursive and then to an iterative algorithm.

```
CALCULATOR( $n$ ):





```

Recall however that besides the optimum value, we are asked to output an optimum sequence of operations. To do this, let us notice that we can find the last operation as follows:

- it is “+1”, if  $\text{calculator}(n) = 1 + \text{calculator}(n - 1)$ ;
- it is “×2”, if  $n$  is divisible by 2 and  $\text{calculator}(n) = 1 + \text{calculator}(n/2)$ ;
- it is “×3”, if  $n$  is divisible by 3 and  $\text{calculator}(n) = 1 + \text{calculator}(n/3)$ .

This allows us to uncover an optimum sequence as follows:

1. find the last operation;
2. replace  $n$  by  $n - 1$ ,  $n/2$ , or  $n/3$  (depending on which of the three cases above happens);
3. iterate (while  $n > 1$ ).

```

CALCULATOR( $n$ ):

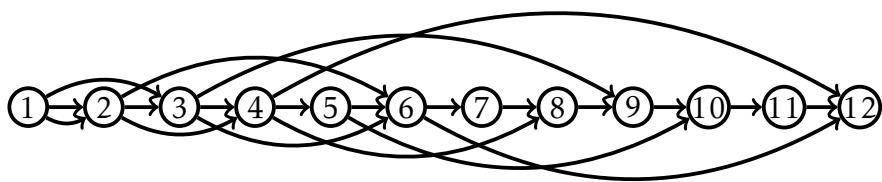




```

The running time of the algorithm is  $O(n)$ .

The algorithm finds implicitly a shortest path in a DAG like the one below.



## Code

```

def calculator(n):
    table = [float('inf')] * (n + 1)

```

```
table[1] = 0

for k in range(2, n + 1):
    table[k] = 1 + table[k - 1]
    if k % 2 == 0:
        table[k] = min(table[k], 1 + table[k // 2])
    if k % 3 == 0:
        table[k] = min(table[k], 1 + table[k // 3])

operations = []
while n > 1:
    operations.append(n)
    if table[n] == 1 + table[n - 1]:
        n = n - 1
    elif n % 2 == 0 and table[n] == 1 + table[n // 2]:
        n = n // 2
    elif n % 3 == 0 and table[n] == 1 + table[n // 3]:
        n = n // 3

return [1] + list(reversed(operations))

if __name__ == '__main__':
    n = int(input())
    sequence = calculator(n)
    print(len(sequence) - 1)
    print(*sequence)
```

### 7.2.3 Edit Distance

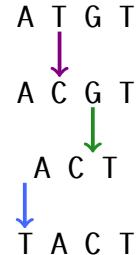
---

#### Edit Distance Problem

Compute the edit distance between two strings.

**Input:** Two strings.

**Output:** The minimum number of single-symbol insertions, deletions, and substitutions to transform one string into the other one.



---

The Edit Distance Problem has many applications in computational biology, natural language processing, spell checking, and many other areas. For example, biologists often compute edit distances when they search for disease-causing mutations.

the edit distance between two strings is defined as the minimum number of single-symbol **insertions**, **deletions**, and **substitutions** to transform one string into the other one.

**Input format.** Two strings consisting of lower case Latin letters, each on a separate line.

**Output format.** The edit distance between them.

**Constraints.** The length of both strings is at least 1 and at most 100.

#### Sample 1.

Input:

```
short  
ports
```

Output:

```
3
```

The second string can be obtained from the first one by deleting s, substituting h for p, and inserting s. This can be compactly visualized by the following *alignment*.

s	h	o	r	t	-
-	p	o	r	t	s

### Sample 2.

Input:

editing  
distance

Output:

5

Delete e, insert s after i, substitute i for a, substitute g for c, insert e to the end.

e	d	i	-	t	i	n	-	g
-	d	i	s	t	a	n	c	e

### Sample 3.

Input:

ab  
ab

Output:

0

**Programming tips:** be careful with recursion ([LD4](#)), use row-by-row iteration through a matrix ([LR3](#)).

### Solution

An *alignment* of two strings is a two-row matrix such that the first (second) row contains the ordered symbols of the first (second) string, interspersed with the space (“-”) symbols in such a way that no two space symbols appear in the same column.

**Exercise Break.** Compute the number of different alignments of two strings of lengths  $n$  and  $m$ .

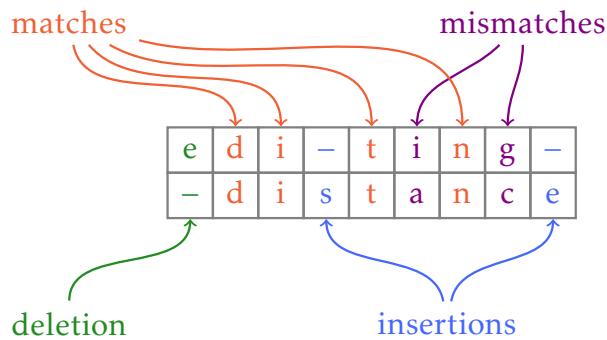
We classify the columns of an alignment as follows:

- a column with a symbol and a space is a **deletion**;
- a column with a gap and a symbol is an **insertion**;

- a column with two equal symbols is a **match**;
- a column with two different symbols is a **mismatch**.

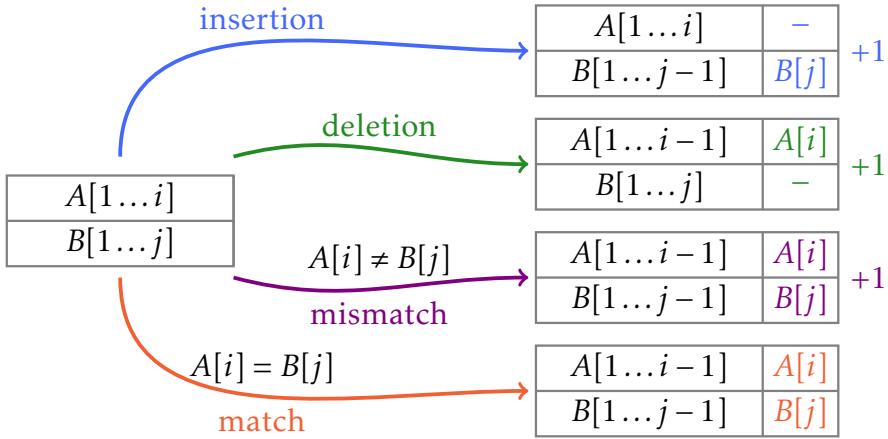
An alignment is *optimal* if it minimizes the total number of mismatches, deletions, and insertions among all possible alignments.

**Exercise Break.** Prove that the Edit Distance Problem can be reduced to finding an optimal alignment of two strings.



In the example above, the last column represents an insertion. By dropping this column, we get an optimal alignment of the first string and a prefix of the second string. Here is an idea — let's compute the edit distance between each pair of prefixes of two strings.

Given strings  $A[1 \dots n]$  and  $B[1 \dots m]$ , we consider their prefixes  $A[1 \dots i]$  and  $B[1 \dots j]$  of length  $i$  and  $j$  and denote the edit distance between them as  $EditDistance(i, j)$ . The last column of an optimal alignment of  $A[1 \dots i]$  and  $B[1 \dots j]$  is either an **insertion**, a **deletion**, a **mismatch**, or a **match**.



Then,

$$EditDistance(i, j) = \min \begin{cases} EditDistance(i, j - 1) + 1 \\ EditDistance(i - 1, j) + 1 \\ EditDistance(i - 1, j - 1) + 1 & \text{if } A[i] \neq B[j] \\ EditDistance(i - 1, j - 1) & \text{if } A[i] = B[j] \end{cases}$$

The base case for this recurrence relation is  $i = 0$  and  $j = 0$ :

$$EditDistance(0, j) = j \quad \text{and} \quad EditDistance(i, 0) = i.$$

This can be stated more compactly as follows: if  $i = 0$  or  $j = 0$ , then

$$EditDistance(i, j) = \max\{i, j\}.$$

The pseudocode below converts this recurrence relation into a recursive algorithm and uses memoization to avoid computing the same thing again.

```





```

The running time of this algorithm is  $O(nm)$  as there are at most  $nm$  recursive calls that are not just  $table$  lookups.

#### 7.2.3.1 Iterative Algorithm

The recursive algorithm computes  $EditDistance(i, j)$  for all  $0 \leq i \leq n$  and  $0 \leq j \leq m$ . One can turn the recursive algorithm into an iterative one that stores the solutions to all subproblems in a two-dimensional table. We fill in the table by traversing it row-by-row. This ensures that by the time we compute the value of  $(i, j)$  cell, the values of the cells  $(i, j - 1)$ ,  $(i - 1, j)$  and  $(i - 1, j - 1)$  are already computed.

```

EDITDISTANCE( $A[1 \dots n], B[1 \dots m]$ ) :





```

The resulting table for our working example is shown in Figure 7.1. The value in each cell is computed from the values of its top, left, and top-left neighbors. For each cell, the incoming arrows indicate one or more out of four cases (`insertion`, `deletion`, `mismatch`, or `match`) that lead to the value of this cell.

The table corresponds to a DAG where all edges have length 1 except for red edges that correspond to matching symbols and have length 0. The algorithm finds a shortest path in this graph from the top left node to the bottom right node.

The running time of the algorithm is  $O(nm)$ . The algorithm uses  $O(nm)$  space to store the two-dimensional array `table`. One can reduce the space consumption to  $O(m)$  (and even  $O(\min\{n, m\})$ ) by noticing that when filling the current row of the table, one only needs the cells from the current row and the previous one. Thus, instead of storing the whole table, it is sufficient to store the current and the previous rows.

**Stop and Think.** You have now computed the edit distance between “editing” and “distance”. But how would you find the five operations to transform “editing” into “distance”?

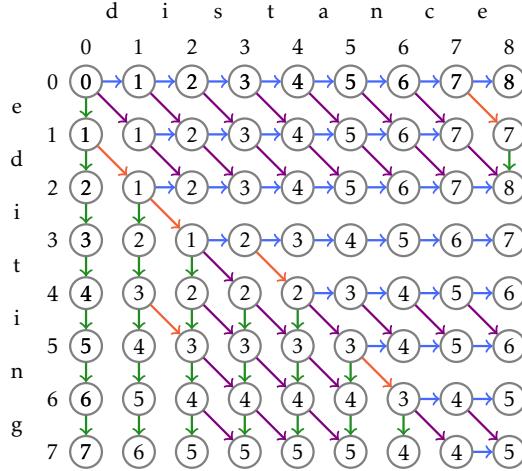


Figure 7.1: The values of a two-dimensional array *table* after the call to `EDITDISTANCE(editing, distance)`.

### 7.2.3.2 Reconstructing an Optimal Alignment

Any path from  $(0, 0)$  to  $(n, m)$  in Figure 7.1 spells an optimal alignment of strings  $A$  and  $B$ .

**Exercise Break.** The path shown in Figure 7.2 corresponds to an optimal alignment of “editing” and “distance”. How many insertions, deletions, matches, and mismatches does this alignment have? Construct the optimal alignment corresponding to this path.

To construct an optimal alignment we will use the backtracking pointers. Looking at Figure 7.1, we see that there are two possibilities to arrive to the bottom right cell: through either a purple or a blue pointer. This means that there exists an optimal alignment whose last column is a **mismatch** and an optimal alignment whose last column is an **insertion**. Let’s consider a **mismatch**. This way, we reconstruct the last column of an optimal alignment:

g
e

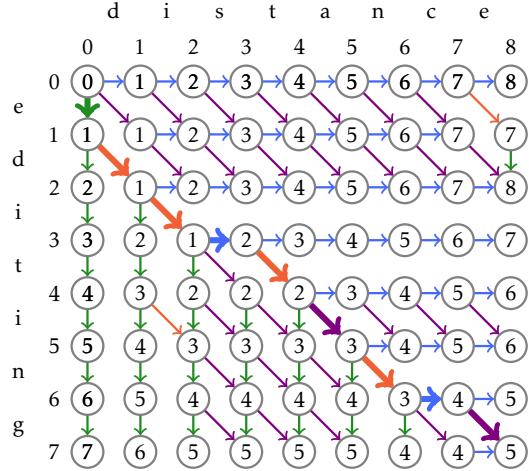


Figure 7.2: A bold path corresponds to an optimal alignment.

Then, we move to the node  $(6,7)$  and continue in a similar fashion. The reconstructed alignment looks as follows.

e	d	i	-	t	i	n	-	g
-	d	i	s	t	a	n	c	e

As this example illustrates, one can uncover an optimal alignment by traversing the arrows back from the bottom-right corner along any path leading to the upper-left corner.

The resulting pseudocode for reconstructing an alignment is shown below. Its running time is  $O(n + m)$ .

```

OUTPUTALIGNMENT( $i, j$ ):
if  $i = 0$  and  $j = 0$ :
    return
if  $i > 0$  and  $\text{table}(i, j) = \text{table}(i - 1, j) + 1$ :
    OUTPUTALIGNMENT( $i - 1, j$ )
    print  $\begin{array}{c} A[i] \\ - \end{array}$ 
else if  $j > 0$  and  $\text{table}(i, j) = \text{table}(i, j - 1) + 1$ :
    OUTPUTALIGNMENT( $i, j - 1$ )
    print  $\begin{array}{c} - \\ B[j] \end{array}$ 
else:
    OUTPUTALIGNMENT( $i - 1, j - 1$ )
    print  $\begin{array}{c} A[i] \\ B[j] \end{array}$ 

```

The algorithm above uses the entire 2d array  $\text{table}$  of size  $(n+1) \times (m+1)$  for reconstructing a solution. As we've discussed previously, the edit distance can be computed with only  $O(m)$  space. However, computing the edit distance together with an optimal alignment in time  $O(nm)$  and space  $O(m)$  is not that easy. See Lesson 5.13 (available [online](#)) in "Bioinformatics Algorithms: an Active Learning Approach" by Compeau and Pevzner. The corresponding algorithm is a neat combination of divide-and-conquer and dynamic programming techniques.

## Code

```

from functools import lru_cache

@lru_cache(maxsize=10000)
def edit_distance(a, b, i, j):
    if i == 0 or j == 0:
        return max(i, j)

    return min(
        edit_distance(a, b, i, j - 1) + 1,

```

```

        edit_distance(a, b, i - 1, j) + 1,
        edit_distance(a, b, i - 1, j - 1) +
            (a[i - 1] != b[j - 1]))
    )

if __name__ == "__main__":
    a, b = input(), input()
    print(edit_distance(a, b, len(a), len(b)))

```

```

def edit_distance(a, b):
    table = [[0] * (len(b) + 1) for _ in range(len(a) + 1)]

    for i in range(len(a) + 1):
        for j in range(len(b) + 1):
            if i == 0 or j == 0:
                table[i][j] = max(i, j)
            else:
                table[i][j] = min(
                    table[i][j - 1] + 1,
                    table[i - 1][j] + 1,
                    table[i - 1][j - 1] +
                        (a[i - 1] != b[j - 1]))
    )

    return table[len(a)][len(b)]

if __name__ == "__main__":
    print(edit_distance(input(), input()))

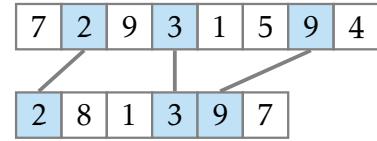
```

## 7.2.4 Longest Common Subsequence of Two Sequences

---

### Longest Common Subsequence of Two Sequences Problem

Compute the maximum length of a common subsequence of two sequences.



**Input:** Two sequences.

**Output:** The maximum length of a common subsequence.

---

Given two sequences  $A = (a_1, a_2, \dots, a_n)$  and  $B = (b_1, b_2, \dots, b_m)$ , their common subsequence of length  $p$  is a set of  $p$  indices

$$\begin{aligned} 1 \leq i_1 < i_2 < \dots < i_p \leq n, \\ 1 \leq j_1 < j_2 < \dots < j_p \leq m. \end{aligned}$$

such that

$$\begin{aligned} a_{i_1} &= b_{j_1}, \\ a_{i_2} &= b_{j_2}, \\ &\vdots \\ a_{i_p} &= b_{j_p}. \end{aligned}$$

The longest common subsequence is a common subsequence of the maximal length among all subsequences.

The problem has applications in data comparison (e.g., diff utility, merge operation in various version control systems), bioinformatics (finding similarities between genes in various species), and others.

**Input format.** First line:  $n$ . Second line:  $a_1, a_2, \dots, a_n$ . Third line:  $m$ . Fourth line:  $b_1, b_2, \dots, b_m$ .

**Output format.**  $p$ .

**Constraints.**  $1 \leq n, m \leq 100$ ;  $-10^9 \leq a_i, b_i \leq 10^9$  for all  $i$ .

**Sample 1.**

Input:

```
3  
2 7 5  
2  
2 5
```

Output:

```
2
```

A common subsequence of length 2 is (2, 5).

**Sample 2.**

Input:

```
1  
7  
4  
1 2 3 4
```

Output:

```
0
```

The two sequences do not share elements.

**Sample 3.**

Input:

```
4  
2 7 8 3  
4  
5 2 8 7
```

Output:

```
2
```

One common subsequence is (2, 7). Another one is (2, 8).

**Programming tips:** use row-by-row iteration through a matrix ([LR3](#)).

## Solution

Consider a longest common subsequence  $C = (c_1, \dots, c_p)$  specified by indices  $1 \leq i_1 < i_2 < \dots < i_p \leq n$  and  $1 \leq j_1 < j_2 < \dots < j_p \leq m$  (thus, for every  $1 \leq q \leq p$ ,  $a_{i_q} = b_{j_q} = c_q$ ).

- Last symbols of  $A$  and  $B$  appear in  $C$ . In this case,  $i_p = n$  and  $j_p = m$ . Then,  $(c_1, \dots, c_{p-1})$  is a longest common subsequence of  $(a_1, \dots, a_{n-1})$  and  $(b_1, \dots, b_{m-1})$ .
  - At least one of the last symbols of  $A$  and  $B$  doesn't appear in  $C$ . In this case, either  $i_p < n$  or  $j_p < m$ . Then,  $(c_1, \dots, c_{p-1})$  lies entirely in either  $(a_1, \dots, a_{n-1})$  or  $(b_1, \dots, b_{m-1})$ .

This way, we reduce the problem for the initial strings  $A$  and  $B$  to the same problem on their prefixes. This motivates the following definition:  $LCS(i, j)$  is the length of the longest common subsequence of  $A[1 \dots i]$  and  $B[1 \dots j]$ . The discussion above implies that this function satisfies the following recurrence relation:

$$LCS(i, j) = \max \begin{cases} LCS(i - 1, j) \\ LCS(i, j - 1) \\ LCS(i - 1, j - 1) + 1 & \text{if } A[i] \equiv B[j] \end{cases}$$

The base case for this recurrence relation is  $i = 0$  or  $j = 0$ :

$$LCS(0, j) = LCS(i, 0) = 0.$$

The resulting algorithm is shown below. Its running time is  $O(nm)$ .

```

LCS(A[1...n], B[1...m]) :


|             |              |                                                 |
|-------------|--------------|-------------------------------------------------|
| table       | $\leftarrow$ | 2d array of size $(n + 1) \times (m + 1)$       |
| table[i][0] | $\leftarrow$ | 0 and table[0][j] $\leftarrow$ 0 for all $i, j$ |
| for         | $i$          | from 1 to $n$ :                                 |
| for         | $j$          | from 1 to $m$ :                                 |
| table[i][j] | $\leftarrow$ | table[i - 1][j]                                 |
| table[i][j] | $\leftarrow$ | max(table[i][j], table[i][j - 1])               |
| if          | A[i] = B[j]: |                                                 |
| table[i][j] | $\leftarrow$ | max(table[i][j], table[i - 1][j - 1] + 1)       |
| return      | table[n][m]  |                                                 |


```

**Stop and Think.** Can you reduce the Longest Common Subsequence Problem to the Edit Distance Problem?

Hint: The longest common subsequence of  $A = (7, 2, 9, 3, 1, 5, 9, 4)$  and  $B = (2, 8, 1, 3, 9, 7)$  is obtained by deleting some symbols from both  $A$  and  $B$ .

7	2	9	-	3	1	5	9	4	-
-	2	-	8	3	-	-	9	-	7

Thus, the Longest Common Subsequence Problem is merely the Edit Distance Problem with the banned “substitution” operations.

## Code

```
def lcs2(first, second):
    table = [[0] * (len(second) + 1)
              for _ in range(len(first) + 1)]

    for i in range(1, len(first) + 1):
        for j in range(1, len(second) + 1):
            table[i][j] = table[i - 1][j]
            table[i][j] = max(table[i][j], table[i][j - 1])
            if first[i - 1] == second[j - 1]:
                table[i][j] = max(table[i][j],
                                   table[i - 1][j - 1] + 1)

    return table[len(first)][len(second)]


if __name__ == '__main__':
    a = int(input()), list(map(int, input().split()))
    b = int(input()), list(map(int, input().split()))
    print(lcs2(a, b))
```

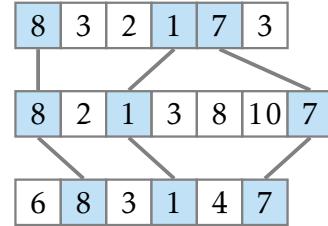
## 7.2.5 Longest Common Subsequence of Three Sequences

### Longest Common Subsequence of Three Sequences Problem

*Compute the maximum length of a common subsequence of three sequences.*

**Input:** Three sequences.

**Output:** The maximum length of a common subsequence.



Given three sequences  $A = (a_1, a_2, \dots, a_n)$ ,  $B = (b_1, b_2, \dots, b_m)$ , and  $C = (c_1, c_2, \dots, c_l)$ , find the length of their longest common subsequence, i.e., the largest non-negative integer  $p$  such that there exist indices

$$\begin{aligned} 1 &\leq i_1 < i_2 < \dots < i_p \leq n, \\ 1 &\leq j_1 < j_2 < \dots < j_p \leq m, \\ 1 &\leq k_1 < k_2 < \dots < k_p \leq l \end{aligned}$$

such that

$$\begin{aligned} a_{i_1} &= b_{j_1} = c_{k_1}, \\ a_{i_2} &= b_{j_2} = c_{k_2}, \\ &\vdots \\ a_{i_p} &= b_{j_p} = c_{k_p}. \end{aligned}$$

**Input format.** First line:  $n$ . Second line:  $a_1, a_2, \dots, a_n$ . Third line:  $m$ . Fourth line:  $b_1, b_2, \dots, b_m$ . Fifth line:  $l$ . Sixth line:  $c_1, c_2, \dots, c_l$ .

**Output format.**  $p$ .

**Constraints.**  $1 \leq n, m, l \leq 100$ ;  $-10^9 \leq a_i, b_i, c_i \leq 10^9$ .

**Sample 1.**

Input:

```
3
1 2 3
3
2 1 3
3
1 3 5
```

Output:

```
2
```

A common subsequence of length 2 is (1,3).

**Sample 2.**

Input:

```
5
8 3 2 1 7
7
8 2 1 3 8 10 7
6
6 8 3 1 4 7
```

Output:

```
3
```

One common subsequence of length 3 in this case is (8,3,7). Another one is (8,1,7).

**Solution**

Let  $LCS(i, j, k)$  be the maximum length of a common subsequence of  $A[1 \dots i]$ ,  $B[1 \dots j]$ , and  $C[1 \dots k]$ . Then,

$$LCS(i, j, k) = \max \begin{cases} LCS(i - 1, j, k) \\ LCS(i, j - 1, k) \\ LCS(i, j, k - 1) \\ LCS(i - 1, j - 1, k - 1) + 1 & \text{if } A[i] = B[j] = C[k] \end{cases}$$

Base case:

$$LCS(0, j, k) = LCS(i, 0, k) = LCS(i, j, 0) = 0.$$

The corresponding algorithm has running time  $O(nmk)$ .

## Code

```
def lcs3(first, second, third):
    table = [
        [
            [0 for _ in range(len(third) + 1)]
            for _ in range(len(second) + 1)
        ]
        for _ in range(len(first) + 1)
    ]

    for i in range(1, len(first) + 1):
        for j in range(1, len(second) + 1):
            for k in range(1, len(third) + 1):
                table[i][j][k] = table[i - 1][j][k]
                table[i][j][k] = max(table[i][j][k],
                                      table[i - 1][j][k])
                table[i][j][k] = max(table[i][j][k],
                                      table[i][j - 1][k])
                if first[i - 1] == second[j - 1] == third[k - 1]:
                    table[i][j][k] = max(
                        table[i][j][k],
                        table[i - 1][j - 1][k - 1] + 1
                    )

    return table[len(first)][len(second)][len(third)]


if __name__ == '__main__':
    a = int(input()), list(map(int, input().split()))
    b = int(input()), list(map(int, input().split()))
    c = int(input()), list(map(int, input().split()))
    print(lcs3(a, b, c))
```

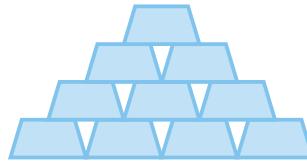
## 7.2.6 Maximum Amount of Gold

### Maximum Amount of Gold Problem

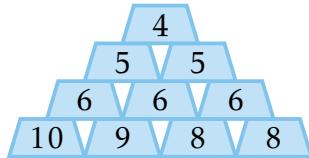
Given a set of gold bars of various weights and a backpack that can hold at most  $W$  pounds, place as much gold as possible into the backpack.

**Input:** A set of  $n$  gold bars of integer weights  $w_1, \dots, w_n$  and a backpack that can hold at most  $W$  pounds.

**Output:** A subset of gold bars of maximum total weight not exceeding  $W$ .



You found a set of gold bars and your goal is to pack as much gold as possible into your backpack that has capacity  $W$ , i.e., it may hold at most  $W$  pounds. There is just one copy of each bar and for each bar you can either take it or not (you cannot take a fraction of a bar). Although all bars appear to be identical in the figure above, their weights vary as illustrated in the figure below.



A natural greedy strategy is to grab the heaviest bar that still fits into the remaining capacity of the backpack and iterate. For the set of bars shown above and a backpack of capacity 20, the greedy algorithm would select gold bars of weights 10 and 9. But an optimal solution, containing bars of weights 4, 6, and 10, has a larger weight!

**Input format.** The first line of the input contains an integer  $W$  (capacity of the backpack) and the number  $n$  of gold bars. The next line contains  $n$  integers  $w_1, \dots, w_n$  defining the weights of the gold bars.

**Output format.** The maximum weight of gold bars that fits into a backpack of capacity  $W$ .

**Constraints.**  $1 \leq W \leq 10^4$ ;  $1 \leq n \leq 300$ ;  $0 \leq w_1, \dots, w_n \leq 10^5$ .

**Sample.**

Input:

```
10 3  
1 4 8
```

Output:

```
9
```

The sum of the weights of the first and the last bar is equal to 9.

### Solution 1: Analyzing the Structure of a Solution

Instead of solving the original problem, we will check whether it is possible to fully pack our backpack with the gold bars: given  $n$  gold bars of weights  $w_0, \dots, w_{n-1}$  (we switched to the 0-based indexing) and an integer  $W$ , is it possible to select a subset of them of total weight  $W$ ?

**Exercise Break.** Show how to use the solution to this problem to solve the Maximum Amount of Gold Problem.

Assume that it is possible to fully pack the backpack: there exists a set  $S \subseteq \{w_0, \dots, w_{n-1}\}$  of total weight  $W$ . Does it include the last bar of weight  $w_{n-1}$ ?

**Case 1:** If  $w_{n-1} \notin S$ , then a backpack of capacity  $W$  can be fully packed using the first  $n - 1$  bars.

**Case 2:** If  $w_{n-1} \in S$ , then we can remove the bar of weight  $w_{n-1}$  from the backpack and the remaining bars will have weight  $W - w_{n-1}$ . Therefore, a backpack of capacity  $W - w_{n-1}$  can be fully packed with the first  $n - 1$  gold bars.

In both cases, we reduced the problem to essentially the same problem with smaller number of items and possibly smaller backpack capacity. We thus consider the variable  $\text{pack}(w, i)$  equal to true if it is possible to fully pack a backpack of capacity  $w$  using the first  $i$  bars, and false, otherwise. The analysis of the two cases above leads to the following recurrence relation for  $i > 0$ ,

$$\text{pack}(w, i) = \text{pack}(w, i - 1) \text{ or } \text{pack}(w - w_{i-1}, i - 1).$$

Note that the second term in the above formula does not make sense when  $w_{i-1} > w$ . Also,  $\text{pack}(0, 0) = \text{true}$ , and  $\text{pack}(0, w) = \text{false}$  for any  $w > 0$ . Overall,

$$\text{pack}(w, i) = \begin{cases} \text{true} & \text{if } i = 0 \text{ and } w = 0 \\ \text{false} & \text{if } i = 0 \text{ and } w > 0 \\ \text{pack}(w, i - 1) & \text{if } i > 0 \text{ and } w_{i-1} > w \\ \text{pack}(w, i - 1) \text{ or } \text{pack}(w - w_{i-1}, i - 1) & \text{otherwise} \end{cases}$$

As  $i$  ranges from 0 to  $n$  and  $w$  ranges from 0 to  $W$ , we have  $O(nW)$  variables. Since  $\text{pack}(\cdot, i)$  depends on  $\text{pack}(\cdot, i-1)$ , we process all variables in the increasing order of  $i$ . In the pseudocode below, we use a two-dimensional array  $\text{pack}$  of size  $(W + 1) \times (n + 1)$ :  $\text{pack}[w][i]$  stores the value of  $\text{pack}(w, i)$ . The running time of this solution is  $O(nW)$ .

```

KNAPSACK([ $w_0, \dots, w_{n-1}$ ],  $W$ ):
 $\text{pack} \leftarrow$  two-dimensional array of size  $(W + 1) \times (n + 1)$ 
initialize all elements of  $\text{pack}$  to false
 $\text{pack}[0][0] \leftarrow \text{true}$ 
for  $i$  from 1 to  $n$ :
    for  $w$  from 0 to  $W$ :
        if  $w_{i-1} > w$ :
             $\text{pack}[w][i] \leftarrow \text{pack}[w][i - 1]$ 
        else:
             $\text{pack}[w][i] \leftarrow \text{pack}[w][i - 1] \text{ OR } \text{pack}[w - w_{i-1}][i - 1]$ 
return  $\text{pack}[W][n]$ 
```

The two-dimensional table below presents the results of the call to  $\text{KNAPSACK}([1, 3, 4], 8)$  and uses F and T to denote false and true values.

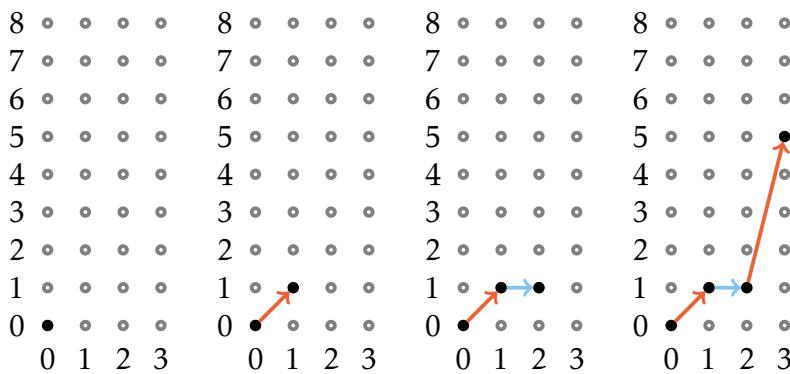
	0	1	2	3
0	T	T	T	T
1	F	T	T	T
2	F	F	F	F
3	F	F	T	T
4	F	F	T	T
5	F	F	F	T
6	F	F	F	F
7	F	F	F	T
8	F	F	F	T

## Solution 2: Analyzing All Subsets of Bars

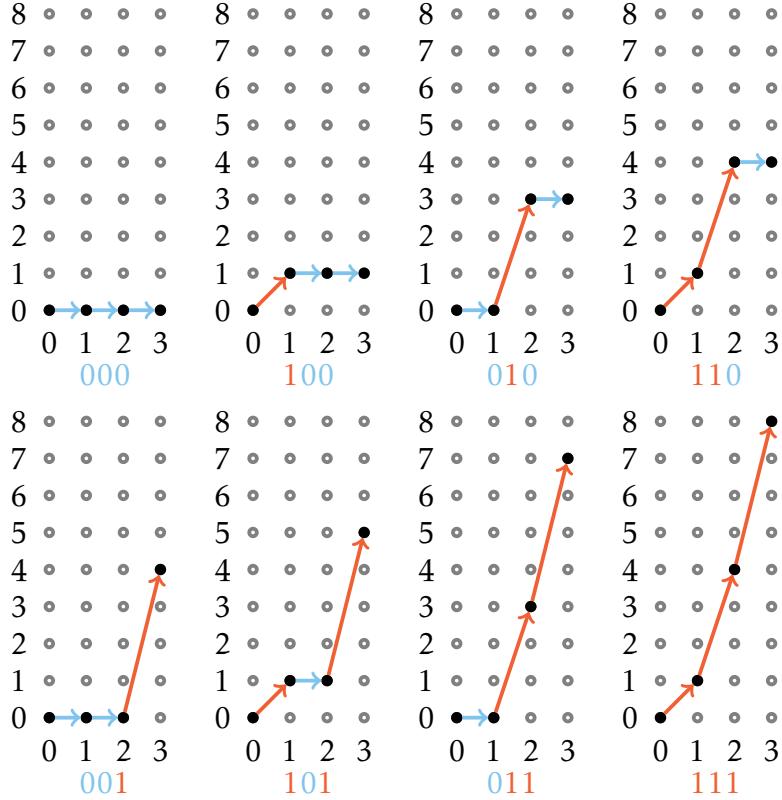
Our goal is to find a subset of  $n$  bars of total weight  $W$ . A straightforward approach to this problem is to go through all subsets and check whether the weight of one of them is equal to  $W$ . Since each bar can be either **skipped** or **taken**, each subset of three bars that we analyzed in the previous section ( $w_0 = 1, w_1 = 3, w_2 = 4$ ) can be represented by a blue-red binary vector:

subset	vector	weight
	000	0
	100	1
	010	3
	110	4
	001	4
	101	5
	011	7
	111	8

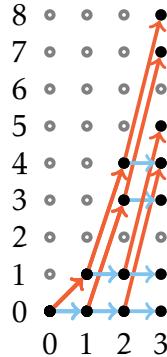
We will now represent each subset of bars as a path starting at node  $(0, 0)$  of a  $(n + 1) \times (W + 1)$  grid. If the first bit is blue, it corresponds to a blue horizontal segment in the grid connecting  $(0, 0)$  with  $(0, 1)$ . If the first bit is red, it corresponds to a red segment in the grid connecting  $(0, 0)$  with  $(1, w_0)$ . After processing the first  $i$  bits, we will have a blue-red path from  $(0, 0)$  to some node  $(i, w)$  of the grid. If the next bit is blue, we will connect  $(i, w)$  with  $(i + 1, w)$ . If the next bit is red, we will connect  $(i, w)$  with  $(i + 1, w + w_i)$  as shown below for the vector **101**:



The Figure below shows the paths corresponding to all eight binary vectors of length 3.



We now superimpose all these eight paths on a single grid:



We classify a node  $(i, w)$  in the grid as true if there is a path from  $(0, 0)$  to  $(i, w)$  in the Figure above, and false otherwise. We can fully pack a knapsack

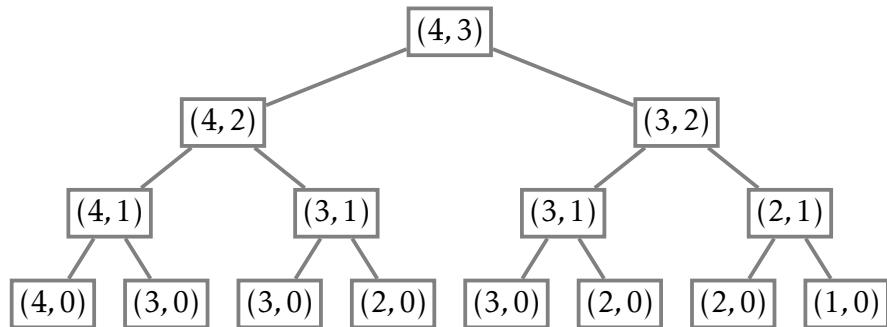
of capacity  $w$  by a subset of the first  $i$  bars if the node  $(i, w)$  is true. A node is true if there is either a blue edge or a red edge into this node, i.e., if either  $(i - 1, w)$  or  $(i - 1, w - w_{i-1})$  are true. This observation brings us to the previous recurrence relation and the same dynamic programming solution.

### Solution 3: Memoization

The following pseudocode recursively computes the recurrence relation from the Solution 1:

```
RECURSIVEKNAPSACK([ $w_0, \dots, w_{n-1}$ ],  $w, i$ ):
if  $i = 0$  and  $w = 0$ :
    return true
else if  $i = 0$  and  $w > 0$ :
    return false
else if  $i > 0$  and  $w_{i-1} > w$ :
    return RECURSIVEKNAPSACK([ $w_0, \dots, w_{n-1}$ ],  $w, i - 1$ )
else:
    return RECURSIVEKNAPSACK([ $w_0, \dots, w_{n-1}$ ],  $w, i - 1$ ) OR
        RECURSIVEKNAPSACK([ $w_0, \dots, w_{n-1}$ ],  $w - w_{i-1}, i - 1$ )
```

A call to  $\text{RECURSIVEKNAPSACK}([w_0, \dots, w_{n-1}], W, n)$  solves the problem but is too slow since it recomputes the same values over and over again. To illustrate this, consider a bag of capacity  $W = 4$  and  $n = 3$  bars of weights  $w_0 = 1, w_2 = 1, w_3 = 1$ . A call  $\text{RECURSIVEKNAPSACK}([1, 1, 1], 4, 3)$  gives rise to the recursion tree shown below (each node shows the values of  $(w, i)$ ). Even for this toy example, the value for  $(w, i) = (3, 1)$  is computed twice. For 20 bars, the recursive tree may become gigantic with the same value computing millions of times.



In order to avoid this recursive explosion, we “wrap” our code with *memoization* using an associative array *pack* which is initially empty. An associative array is an abstract data type that stores *(key, value)* pairs. It is supported by many programming languages and is usually implemented as a hash table or a search tree (in C++ and Java it is called a *map*, in Python it is called a *dictionary*). In the implementation below, an associative array *pack* is used to store the Boolean values for *(w, i)* pairs.

```
MEMOIZEDKNAPSACK([ $w_0, \dots, w_{n-1}$ ], pack,  $w, i$ ):
if ( $w, i$ ) is not in pack:
    if  $i = 0$  and  $w = 0$ :
        pack[ $(w, i)$ ]  $\leftarrow$  true
    else if  $i = 0$  and  $w > 0$ :
        pack[ $(w, i)$ ]  $\leftarrow$  false
    else if  $i > 0$  and  $w_{i-1} > w$ :
        pack[ $(w, i)$ ]  $\leftarrow$  MEMOIZEDKNAPSACK([ $w_0, \dots, w_{n-1}$ ], pack,  $w, i - 1$ )
    else:
        pack[ $(w, i)$ ]  $\leftarrow$  MEMOIZEDKNAPSACK([ $w_0, \dots, w_{n-1}$ ], pack,  $w, i - 1$ ) OR
                           MEMOIZEDKNAPSACK([ $w_0, \dots, w_{n-1}$ ], pack,  $w - w_{i-1}, i - 1$ )
return pack[ $(w, i)$ ]
```

The running time of the resulting solution is  $O(nW)$  since there are at most that many recursive calls that are not just lookups in the associative array. Thus, the running time is the same as of the corresponding iterative algorithm. In practice, iterative solution are usually faster since they have no recursion overhead and use simpler data structures (e.g., an array instead of a hash table). For the *considered problem*, however, the situation is different: for *some* datasets, the recursive version is faster than the iterative one. For example, if we multiply all the weights by 10, then the running time of the iterative algorithm is also multiplied by 10 while the running time of the recursive algorithms stays essentially the same. In general, if all possible subproblems need to be solved, then the iterative version is usually faster.

## Code

```

from sys import stdin

def knapsack(weights, capacity):
    n = len(weights)
    pack = [[False] * (n + 1) for _ in range(capacity + 1)]
    pack[0][0] = True

    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] > w:
                pack[w][i] = pack[w][i - 1]
            else:
                pack[w][i] = pack[w][i - 1] or \
                    pack[w - weights[i - 1]][i - 1]

    for s in range(capacity, -1, -1):
        if pack[s][n]:
            return s

if __name__ == '__main__':
    input_capacity, input_n, *input_weights = \
        list(map(int, stdin.read().split()))
    assert len(input_weights) == input_n
    print(knapsack(input_weights, input_capacity))

```

```

from sys import stdin

def memoized(weights, pack, w, i):
    if (w, i) not in pack:
        if i == 0 and w == 0:
            pack[(w, i)] = True
        elif i == 0 and w > 0:
            pack[(w, i)] = False

```

```
        elif i > 0 and weights[i - 1] > w:
            pack[(w, i)] = memoized(weights, pack, w, i - 1)
        else:
            pack[(w, i)] = memoized(weights, pack, w, i - 1) or \
                            memoized(weights, pack,
                                     w - weights[i - 1], i - 1)

    return pack[(w, i)]


if __name__ == '__main__':
    input_capacity, input_n, *input_weights = \
        list(map(int, stdin.read().split()))
    assert len(input_weights) == input_n
    memorized_pack = {}
    for s in range(input_capacity, -1, -1):
        if memoized(input_weights, memorized_pack, s, input_n):
            print(s)
            break
```

### 7.2.7 Splitting the Pirate Loot

---

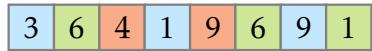
#### 3-Partition Problem

Partition a set of integers into three subsets with equal sums.

**Input:** A sequence of integers

$v_1, v_2, \dots, v_n$ .

**Output:** Check whether it is possible to partition them into three subsets with equal sums, i.e., check whether there exist three disjoint sets  $S_1, S_2, S_3 \subseteq \{1, 2, \dots, n\}$  such that  $S_1 \cup S_2 \cup S_3 = \{1, 2, \dots, n\}$  and



$$\sum_{i \in S_1} v_i = \sum_{j \in S_2} v_j = \sum_{k \in S_3} v_k.$$

---

Three pirates are splitting their loot consisting of  $n$  items of varying value. Can you help them to evenly split the loot?

**Input format.** The first line contains an integer  $n$ . The second line contains integers  $v_1, v_2, \dots, v_n$  separated by spaces.

**Output format.** Output 1, if it possible to partition  $v_1, v_2, \dots, v_n$  into three subsets with equal sums, and 0 otherwise.

**Constraints.**  $1 \leq n \leq 20$ ,  $1 \leq v_i \leq 30$  for all  $i$ .

**Sample 1.**

Input:

```
4
3 3 3 3
```

Output:

```
0
```

### Sample 2.

Input:

```
1  
30
```

Output:

```
0
```

### Sample 3.

Input:

```
13  
1 2 3 4 5 5 7 7 8 10 12 19 25
```

Output:

```
1
```

$$1 + 3 + 7 + 25 = 2 + 4 + 5 + 7 + 8 + 10 = 5 + 12 + 19.$$

### Solution

Let's denote  $v_1 + v_2 + \dots + v_i$  as  $\text{sum}(i)$ . Splitting the set of  $n$  items evenly to three parts is possible only if their total value is divisible by three, i.e.,  $\text{sum}(n) = 3V$ , where  $V$  is an integer. Then, we need to partition  $n$  numbers into three parts where the sum of numbers in each part is equal to  $V$ .

Instead of partitioning all  $n$  items, let's try to solve a "smaller" problem of partitioning the first  $i$  items into parts of value  $s_1$ ,  $s_2$ , and  $\text{sum}(i) - s_1 - s_2$ . We define  $\text{split}(i, s_1, s_2) = \text{true}$  if such partition is possible (and  $\text{false}$ , otherwise) and note that the pirates can fairly split the loot only if  $\text{split}(n, V, V) = \text{true}$ .

**Stop and Think.** Given the first five items

3	6	4	1	9
---	---	---	---	---

$\text{split}(5, 4, 13) = \text{true}$ . Find all other values  $\text{split}(5, s_1, s_2)$  equal to  $\text{true}$ .

**Stop and Think.** Imagine that you have already constructed the binary two-dimensional array  $\text{split}(i - 1, s_1, s_2)$  for all possible values  $0 \leq s_1 \leq V$  and  $0 \leq s_2 \leq V$ . Can you use this array to construct the array  $\text{split}(i, s_1, s_2)$ ?

Assume that  $\text{split}(i, s_1, s_2) = \text{true}$ , that is, one can split the first  $i$  numbers into three parts such that the sum of numbers in the first part is  $s_1$  and the sum of numbers in the second part is  $s_2$ .

1. The  $i$ -th number belongs to the first part. Then  $v_i \leq s_1$ . By taking it out of the first part, we will partition the first  $i - 1$  numbers into three parts such that the sum of the first two parts is  $s_1 - v_i$  and  $s_2$ , that is,  $\text{split}(i - 1, s_1 - v_i, s_2) = \text{true}$ .
2. The  $i$ -th number belongs to the second part. Then  $v_i \leq s_1$ . Similarly to the previous case,  $v_i \leq s_2$  and  $\text{split}(i - 1, s_1, s_2 - v_i) = \text{true}$ .
3. The  $i$ -th item belongs to the third part. Then,  $\text{split}(i - 1, s_1, s_2) = \text{true}$ .

This way, one computes the value of  $\text{split}(i, s_1, s_2)$  by looking at

$$\text{split}(i - 1, s_1 - v_i, s_2), \text{split}(i - 1, s_1, s_2 - v_i), \text{ and } \text{split}(i - 1, s_1, s_2).$$

The base case for this recurrence relation is:  $\text{split}(0, 0, 0) = \text{true}$  and  $\text{split}(0, s_1, s_2) = \text{false}$  if  $s_1 + s_2 > 0$ .

```

SPLIT( $v_1, \dots, v_n$ ):
if  $v_1 + \dots + v_n$  is not divisible by 3:
    return false
 $V \leftarrow (v_1 + \dots + v_n)/3$ 
split  $\leftarrow$  three-dimensional array of size  $(n + 1) \times (V + 1) \times (V + 1)$ 
initialize all elements of split to false
split[0][0][0]  $\leftarrow$  true
for  $i$  from 1 to  $n$ :
    for  $s_1$  from 0 to  $V$ :
        for  $s_2$  from 0 to  $V$ :
            split[i][s1][s2]  $\leftarrow$  split[i - 1][s1][s2]
            if  $s_1 \geq v_i$ :
                split[i][s1][s2]  $\leftarrow$  split[i][s1][s2] OR split[i - 1][s1 - v_i][s2]
            if  $s_2 \geq v_i$ :
                split[i][s1][s2]  $\leftarrow$  split[i][s1][s2] OR split[i - 1][s1][s2 - v_i]
return split[n][V][V]

```

The running time is  $O(nV^2)$ .

## Code

```
from itertools import product

def split(values):
    v, n = sum(values), len(values)
    if v % 3 != 0:
        return False
    v = v // 3

    table = [[[False for _ in range(v + 1)]
              for _ in range(v + 1)] for _ in range(n + 1)]
    table[0][0][0] = True

    for i in range(1, n + 1):
        for s1, s2 in product(range(v + 1), repeat=2):
            table[i][s1][s2] = table[i - 1][s1][s2]
            if s1 >= values[i - 1]:
                table[i][s1][s2] |= table[i - 1][s1 - values[i - 1]][s2]
            if s2 >= values[i - 1]:
                table[i][s1][s2] |= table[i - 1][s1][s2 - values[i - 1]]

    return table[n][v][v]

if __name__ == '__main__':
    n = input()
    values = list(map(int, input().split()))
    print(1 if split(values) else 0)
```

## 7.2.8 Maximum Value of an Arithmetic Expression

### Maximum Value of an Arithmetic Expression Problem

*Parenthesize an arithmetic expression to maximize its value.*

**Input:** An arithmetic expression consisting of digits as well as plus, minus, and multiplication signs.

**Output:** Add parentheses to the expression in order to maximize its value.

$$\begin{array}{c} ((8 - 5) \times 3) = 9 \\ \uparrow \\ 8 - 5 \times 3 \\ \downarrow \\ (8 - (5 \times 3)) = -7 \end{array}$$

For example, for an expression  $(3 + 2 \times 4)$  there are two ways of parenthesizing it:  $(3 + (2 \times 4)) = 11$  and  $((3 + 2) \times 4) = 20$ .

**Exercise Break.** Parenthesize the expression  $(5-8+7\times 4-8+9)$  to maximize its value.

**Input format.** The only line of the input contains a string  $s$  of length  $2n + 1$  for some  $n$ , with symbols  $s_0, s_1, \dots, s_{2n}$ . Each symbol at an even position of  $s$  is a digit (that is, an integer from 0 to 9) while each symbol at an odd position is one of three operations from  $\{+, -, *\}$ .

**Output format.** The maximum value of the given arithmetic expression among all possible orders of applying arithmetic operations.

**Constraints.**  $0 \leq n \leq 14$  (hence the string contains at most 29 symbols).

#### Sample.

Input:

5-8+7\*4-8+9

Output:

200

$200 = (5 - ((8 + 7) \times (4 - (8 + 9))))$

## Solution

Each of the five operations in the expression

$$(5 - 8 + 7 \times 4 - 8 + 9)$$

can be the last (or the outermost) one. Consider the case when the last one is “ $\times$ ” (that is, multiplication). In this case, we need to parenthesize two *subexpressions*

$$(5 - 8 + 7) \text{ and } (4 - 8 + 9)$$

so that the product of their values is maximized. To find this out, we find the minimum and maximum values of these two subexpressions:

$$\begin{aligned}\min(5 - 8 + 7) &= (5 - (8 + 7)) = -10, \\ \max(5 - 8 + 7) &= ((5 - 8) + 7) = 4, \\ \min(4 - 8 + 9) &= (4 - (8 + 9)) = -13, \\ \max(4 - 8 + 9) &= ((4 - 8) + 9) = 5.\end{aligned}$$

From these values, we conclude that the maximum value of the product is 130. Assume that the input dataset is of the form

$$d_0 \quad op_0 \quad d_1 \quad op_1 \quad \cdots \quad op_{n-1} \quad d_n,$$

where each  $d_i$  is a digit and each  $op_j \in \{+, -, \times\}$  is an arithmetic operation. The discussion above suggests that we compute the minimum value and the maximum value of each subexpression of the form

$$E_{l,r} = d_l \quad op_l \quad d_{l+1} \quad op_{l+1} \quad \cdots \quad op_{r-1} \quad d_r,$$

where  $0 \leq l \leq r \leq n$ . Let  $\minValue(l, r)$  and  $\maxValue(l, r)$  be, respectively,

the minimum value and the maximum value of  $E_{l,r}$ . Then,

$$\minValue(l, r) = \min_{l \leq m < r} \begin{cases} \minValue(l, m) & op_m \quad minValue(m + 1, r) \\ \minValue(l, m) & op_m \quad maxValue(m + 1, r) \\ \maxValue(l, m) & op_m \quad minValue(m + 1, r) \\ \maxValue(l, m) & op_m \quad minValue(m + 1, r) \end{cases}$$

$$\maxValue(l, r) = \max_{l \leq m < r} \begin{cases} \minValue(l, m) & op_m \quad minValue(m + 1, r) \\ \minValue(l, m) & op_m \quad maxValue(m + 1, r) \\ \maxValue(l, m) & op_m \quad minValue(m + 1, r) \\ \maxValue(l, m) & op_m \quad minValue(m + 1, r) \end{cases}$$

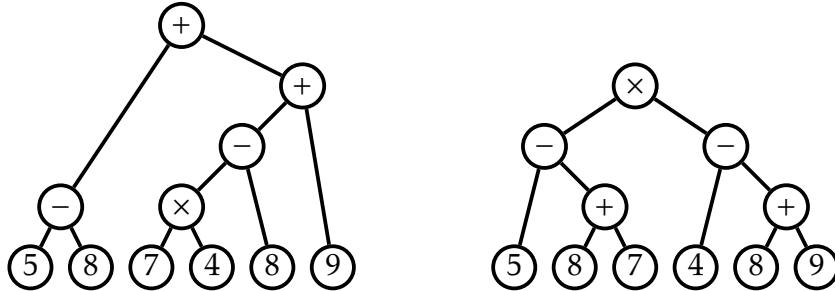
The base case is  $l = r$ :

$$\minValue(l, l) = \maxValue(l, l) = d_l.$$

These two recurrence relations allow us to compute the optimal values for  $E_{l,r}$  by going through all possibilities of breaking  $E_{l,r}$  into two subexpressions  $E_{l,m}$  and  $E_{m+1,r}$ .

Another way of looking at the resulting recurrence relation is the following. A natural way to visualize a particular parenthesizing is to represent it as a binary tree:

$$((5 - 8) + (((7 \times 4) - 8) + 9)) \quad (5 - ((8 + 7) \times (4 - (8 + 9))))$$



Then, our recurrence relation says the following. A tree consists of a root and two subtrees. In order to find an optimum shape of a tree, we go through all possibilities for the root (the parameter  $m$  is responsible for this) and then compose a tree out of two optimal subtrees.

As usual, it is straightforward to convert a recurrence relation into a recursive algorithm. The recursive procedure takes the indices  $l$  and  $r$  as parameters and uses them to compute the minimum and the maximum value of the subexpression  $E_{l,r}$ . Before trying to compute it, it checks whether these two values are already stored in  $\text{table}[l,r]$  where  $\text{table}$  is an associative array needed to store the results that are already computed. If there is no entry  $\text{table}[l,r]$ , the recursive procedure computes the two values using the recurrence relation, stores them in the table, and returns them. The final answer corresponds to  $l = 0$  and  $r = n$ . The running time is  $O(n^3)$ : there are  $O(n^2)$  possible pairs  $(l,r)$ , for each such pair the recursive procedure checks all possible values for  $l \leq m < r$ .

To convert the recursive algorithm into an iterative one, one uses two two-dimensional tables  $\text{mins}[0..n][0..n]$  and  $\text{maxs}[0..n][0..n]$  to store the minimum values and the maximum values of all subexpressions. When filling in these two tables, we should ensure that by the time we compute the optimal values for  $E_{l,r}$ , the optimal values of  $E_{l,m}$  and  $E_{m+1,r}$ , for all  $m$ , are already computed. One possibility to guarantee it is to enumerate all pairs  $(l,r)$  in the order of increasing value of  $r - l$ . This is done by using a parameter  $s = r - l$  in the pseudocode below.

```

MAXVALUE( $d_0 op_0 d_1 op_1 \dots d_n$ ):
  mins, maxs  $\leftarrow$  2d-arrays of size  $(n + 1) \times (n + 1)$ 
  fill mins with  $+\infty$ , fill maxs with  $-\infty$ 
  for  $i$  from 0 to  $n$ :
    mins[ $i$ ][ $i$ ]  $\leftarrow d_i$ , maxs[ $i$ ][ $i$ ]  $\leftarrow d_i$ 
  for  $s$  from 1 to  $n$ :
    for  $l$  from 1 to  $n - s$ :
       $r \leftarrow l + s$ 
      for  $m$  from  $l$  to  $r - 1$ :
         $a \leftarrow \text{mins}[l][m] \quad op_m \quad \text{mins}[m + 1][r]$ 
         $b \leftarrow \text{mins}[l][m] \quad op_m \quad \text{maxs}[m + 1][r]$ 
         $c \leftarrow \text{maxs}[l][m] \quad op_m \quad \text{mins}[m + 1][r]$ 
         $d \leftarrow \text{maxs}[l][m] \quad op_m \quad \text{maxs}[m + 1][r]$ 
         $\text{mins}[l][r] \leftarrow \min(\text{mins}[l][r], a, b, c, d)$ 
         $\text{maxs}[l][r] \leftarrow \max(\text{maxs}[l][r], a, b, c, d)$ 
  return maxs[0][ $n$ ]

```

## Code

```
from functools import lru_cache

def evaluate(left, right, operation):
    if operation == '+':
        return left + right
    elif operation == '-':
        return left - right
    elif operation == '*':
        return left * right
    else:
        assert False

@lru_cache(maxsize=1000)
def min_and_max(dataset, l, r):
    if l == r:
        return int(dataset[2 * l]), int(dataset[2 * l])

    min_value, max_value = float('inf'), -float('inf')

    for m in range(l, r):
        lmin, lmax = min_and_max(dataset, l, m)
        rmin, rmax = min_and_max(dataset, m + 1, r)
        op = dataset[2 * m + 1]

        cases = [
            evaluate(lmin, rmin, op),
            evaluate(lmin, rmax, op),
            evaluate(lmax, rmin, op),
            evaluate(lmax, rmax, op)
        ]

        min_value = min(min_value, min(cases))
        max_value = max(max_value, max(cases))

    return min_value, max_value
```

```

if __name__ == "__main__":
    expression = input()
    n = len(expression) // 2
    print(min_and_max(dataset=expression, l=0, r=n)[1])

```

```

def maximum_value(dataset):
    def evaluate(left, right, operation):
        if operation == '+':
            return left + right
        elif operation == '-':
            return left - right
        elif operation == '*':
            return left * right
        else:
            assert False

    n = len(dataset)
    mins = [
        [float('inf')] for _ in range(n + 1)] for _ in range(n + 1)
    ]
    maxs = [
        [-float('inf')] for _ in range(n + 1)] for _ in range(n + 1)
    ]

    for i in range(0, n + 1, 2):
        mins[i][i + 1] = int(dataset[i])
        maxs[i][i + 1] = int(dataset[i])

    for size in range(1, n + 1, 2):
        for l in range(0, n, 2):
            if l + size > n:
                continue
            r = l + size
            for m in range(l + 1, r, 2):
                oper = dataset[m]
                cases = [

```

```
        evaluate(mins[1][m], mins[m + 1][r], oper),
        evaluate(mins[1][m], maxs[m + 1][r], oper),
        evaluate(maxs[1][m], mins[m + 1][r], oper),
        evaluate(maxs[1][m], maxs[m + 1][r], oper)
    ]
mins[1][r] = min(mins[1][r], min(cases))
maxs[1][r] = max(maxs[1][r], max(cases))

return maxs[0][n]

if __name__ == "__main__":
    print(maximum_value(input())))

```

## 7.3 Designing Dynamic Programming Algorithms

Now when you've seen several dynamic programming algorithms, let's summarize and review the main steps of designing such algorithms.

**Define subproblems.** The first and the most important step is defining subproblems and writing down a recurrence relation (with a base case). This is usually done either by analyzing the structure of an optimal solution or by optimizing a brute force solution.

**Design a recursive algorithm.** Convert a recurrence relation into a recursive algorithm:

- store a solution to each subproblem in a table;
- before solving a subproblem check whether its solution is already stored in the table (memoization).

**Design an iterative algorithm.** Convert a recursive algorithm into an iterative algorithm:

- initialize the table;
- go from smaller subproblems to larger ones.

(Recall that there are cases when a recursive approach is preferable. See a discussion in Section [7.2.6](#).)

**Estimate the running time.** Prove an upper bound on the running time. Usually, the product of the number of subproblems and the time needed to solve a subproblem provides an upper bound on the running time.

**Uncover a solution.** Uncover an optimal solution, by backtracking through the used recurrence relation.

**Save space.** Exploit the regular structure of the table to check whether space can be saved as compared to the straightforward solution.



# Chapter 8: Best Programming Practices (Optimal)

Programming is an art of not making off-by-one errors. In this chapter, we 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, debuggable, and efficient code.

## 8.1 Language Independent

### 8.1.1 Code Format

- LF1 Use code autoformatting of your favorite IDE (integrated development environment) or code editor. For example, many programmers use Eclipse, IDEA, PyCharm, Visual Studio.
- LF2 Do not use spaces and tabs for indentation simultaneously. If the tab size changes, the code will be formatted ugly.

### 8.1.2 Code Structure

- LC1 Structure your code. Structured programming paradigm is aimed at improving the clarity and reducing the development time of your programs by making extensive use of subroutines. Break your code into many subroutines such that each subroutine is responsible for a single task.

In particular, separate reading the input, computing the result, and writing the output. This makes it easier to update your code: e.g., later, the input format may change. This also makes it easier to test your code since computing the result in a separate function simplifies stress testing.

- LC2 Avoid copy-pasting of one section of the code into another section. When copying a piece of code, you copy all its potential bugs. Instead of copying, add a new function (or class) and call it two times with different parameters.

LC3 Make your code compact if it does not reduce its readability. For example, if *condition* is a Boolean variable or expression then use the latter of the following two programs that achieve the same goal:

```
if condition:  
    return true  
else:  
    return false  
  
return condition
```

When computing the minimum number in an array, instead of

```
if current < minimum:  
    minimum ← current
```

use

```
minimum ← min(minimum, current)
```

LC4 The scope of variables should be as small as possible.

LC5 When possible, prefer range-based for loops to index-based for loops. This will reduce the chance of an OBOB. For example, instead of

```
sumOfElements ← 0  
for i from 0 to length(array) – 1:  
    sumOfElements ← sumOfElements + array[i]
```

use

```
sumOfElements ← 0  
for each element in array:  
    sumOfElements ← sumOfElements + element
```

LC6 Do not reinvent the wheel, but be sure to be able to build a wheel if needed. Many classical algorithms and data structures have build-in implementations in your favorite programming language. Examples:

binary search algorithm, sorting algorithms, binary heaps, pattern matching algorithms, and many others. Still, there are many reasons to implement all these algorithms at least once by yourself:

- (a) It is interesting! The corresponding algorithms and data structures are usually based on beautifully simple ideas.
- (b) As we mentioned in the beginning of the book, one of the best ways to understand an algorithm is to implement it.
- (c) A more pragmatic reason: you may be asked about these classical algorithms and data structures at your next technical interview.
- (d) Even if using a built-in implementation as a black box, one needs to know what to expect from this black box when the size of the input data grows.
- (e) Sometimes, one needs to extend a given implementation of an algorithm or a data structure in order to solve a given computational problem. For this, again, one needs to know the details of this implementation.

Hence, if a programming challenge asks you to implement a particular algorithm, then implement it from scratch, even if your favorite programming language contains a built-in implementation. In real life, use a built-in implementation: it is guaranteed to be bug-free and it is already optimized.

### 8.1.3 Names and Comments

LN1 Use meaningful names for variables. The length of a variable name should be proportional to the size of its scope, where the scope is defined as the part of the program where the variable is “visible.” Using a name like *speed* instead of *s* will help your team members to read your program and will help you to debug it. Note that problem statements and mathematical formulas usually contain many single letter variables. This makes it tempting to use the same single letter names when implementing a program. At the same time, the styles of mathematical text and code differ a lot. In particular, formulas are surrounded by text, while code mainly consists of instructions.

A place where a short variable name is acceptable is a counter for a short loop. Still, if there are, say, two nested for loops, do not call the corresponding counters just  $i$  and  $j$  because it is difficult to catch a bug caused by using one of these counters instead of another. For the same reason, do not use variables like `something1` and `something2`: it is easy to make a typo in the name of one of them and to get the other one. Instead, use names like `something_first` and `something_second`.

- LN2 Likewise, use meaningful function names. Use verbs explaining what the function is supposed to do. Each function should be responsible for a single thing. If a natural name for your function contains two verbs (like `ReadFromfileAndSort`), this is a clear indicator to split it into two functions (`ReadFromfile` and `Sort`). If a function returns a Boolean value, name it like `IsEmpty` instead of just `Empty`.
- LN3 Do not use comments to acquit bad variable names. Instead of calling a variable  $n$  and placing a comment like “number of balls”, just call this variable `numberOfBalls`.  
Comments to functions should be placed near their declaration rather than their implementation.  
Avoid commented out code.

#### 8.1.4 Debugging

- LD1 Turn on compiler/interpreter warnings. Although inexperienced programmers sometimes view warnings as a nuisance, they help you to catch some bugs at the early stages of your software implementations.
- LD2 Use assert statements. Each time, there is a condition that must be true at a certain point of your program, add a line

```
assert(condition)
```

A *postcondition* (*precondition*) is a statement that has to be true before (or after) the call to a function. It makes sense to state preconditions

and postconditions for every function in your program. For example, it would save you time if you added a line

```
assert(index1 ≠ index2)
```

when implementing an algorithm for the Maximum Pairwise Product Problem in Section 3.2.2.

The assert statements can also be used to ensure that a certain point in your program is never reached. See the following pseudocode for computing the greatest common divisor of two positive integers.

```
GREATESTCOMMONDIVISOR(a, b):
assert(a ≥ 1 and b ≥ 1)
for d from min(a, b) downto 1:
    if a mod d = 0 and b mod d = 0:
        return d
assert(False)
```

- LD3 Do not optimize your code at early stages. As Donald Knuth once said, premature optimization is the root of all evil in programming. Your code should be correct and should have the expected asymptotic running time. Do not apply any non-asymptotic optimizations before you ensure the correctness of your code. If your code is correct (in particular, it does not get the wrong answer feedback from the grader), but is still too slow (gets time limit exceeded message), start optimizing it. Measure its running time and locate bottlenecks in the code. Note that when compilers apply code optimization, even professional programmers have difficulties predicting bottlenecks.
- LD4 Be careful with recursion. In some cases, recursive solutions are easier to implement and more readable than the corresponding iterative solutions. For example, it is easier to explore graphs recursively rather than iteratively. Also, for many dynamic programming algorithms, it is natural to first implement a recursive approach and then to convert it into an iterative one if needed. As usual, advantages come together with some disadvantages. The main two cons of recursion that should be kept in mind are the following.

**Stack overflow.** If using recursion, make sure to avoid the stack overflow issue that usually occurs due to high recursion depth. Test your program on inputs that force it to make deep recursive calls (say, a graph that is just a path with  $10^5$  nodes). Increase the stack size if needed. (See [PG10](#) on how to do this for Python.)

**Performance.** In many cases (but not always!), an iterative solution is faster and uses less memory than the corresponding recursive one. The reason is that a recursive solution needs to store all function calls and its local variables on stack.

### 8.1.5 Integers and Floating Point Numbers

LI1 Avoid integer overflow. Check the bounds on the input values, estimate the maximum value for the intermediate results and pick a sufficiently large numeric type.

When computing modulo  $m$ , take every intermediate result modulo  $m$ . Say, you want to compute the remainder of the product of all elements of *array* modulo 17. The naive way of doing this is the following.

```
product ← 1
for element in array:
    product ← product · element
return product mod 17
```

In languages with integer overflow (like C++ and Java) this will give a wrong result in many cases: even if *array* has only 100 elements and all its elements are equal to 2, the product does not fit into 64 bits. In languages with out-of-the-box long arithmetic, this code will be slower than needed as *product* is getting larger at every iteration. E.g., in Python, both `pow(a, b, m)` and `pow(a, b) % m` compute  $a^b \bmod m$ , but the former one works faster.

The right way to compute the product is the following.

```
product ← 1
for element in array:
    product ← (product · element) mod 17
return product
```

- LI2 Beware of taking negative numbers modulo  $m$ . In many programming languages,  $(-2)\%5 \neq 3\%5$ . If  $a$  is possibly negative, instead of

```
x ← a mod m
```

use

```
x ← ((a mod m) + m) mod m
```

- LI3 Avoid floating point numbers whenever possible. In the Maximum Value of the Loot Problem (Section 5.2.2) you need to compare

$$\frac{p_i}{w_i} \text{ and } \frac{p_j}{w_j},$$

where  $p_i$  and  $p_j$  ( $w_i$  and  $w_j$ ) are prices (weights) of two compounds. Instead of comparing these *rational* numbers, compare *integers*  $p_i \cdot w_j$  and  $p_j \cdot w_i$ , since integers are faster to compute and precise. However, remember about integer overflow when computing products of large numbers!

In the Closest Points Problem (Section 6.2.7) you need to compare the distances between a pair of points  $(x_1, y_1)$  and  $(x_2, y_2)$  and a pair of points  $(x_3, y_3)$  and  $(x_4, y_4)$ :  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$  and  $\sqrt{(x_3 - x_4)^2 + (y_3 - y_4)^2}$ . Instead of comparing these values, compare the values of their squares:  $(x_1 - x_2)^2 + (y_1 - y_2)^2$  and  $(x_3 - x_4)^2 + (y_3 - y_4)^2$ . In fact, in this problem you need to deal with non-integer numbers just once: only when you output the result.

### 8.1.6 Strings

- LS1 Instead of implementing your own method for checking whether two strings are equal, use a built-in implementation of your favorite programming language: since string comparison is a frequently used operation, a built-in implementation is already optimized.
- LS2 When taking a substring of a string through a built-in method, check whether it creates a new string for this (and hence, uses additional time and space) or not.

### 8.1.7 Ranges

LR1 Use 0-based arrays. Even if the problem statement specifies a 1-based sequence  $a_1, \dots, a_n$ , store it in a 0-based array  $A[0..n - 1]$  (such that  $A[i] = a_{i-1}$ ) instead of a 1-based array  $A[0..n]$  (such that  $A[i] = a_i$ ). In most programming languages, arrays are 0-based. A 0-based array contains only the input data, while an array  $A[0..n]$  contains a dummy element  $A[0]$  that you may accidentally use in your program. For this reason, the size of a 0-based array is equal to the number of input elements making it easier to iterate through it.

To illustrate this point, compare the following two implementations of a function that reads an integer  $n$  followed by reading integers  $a_1, a_2, \dots, a_n$ . Although this section discusses language-independent good programming practices, we take a liberty to present two Python implementations.

The first implementation uses a 1-based array  $A$ .

```
n = int(stdin.readline())
A = [None] * (n + 1)
for i in range(1, n + 1):
    A[i] = int(stdin.readline())
```

The second one uses a 0-based array  $A$ .

```
n = int(stdin.readline())
A = [None] * n
for i in range(len(A)):
    A[i] = int(stdin.readline())
```

The former version has more places for potential OBOBs.

LR2 For recursive implementations, use semiopen intervals to avoid OBOBs. A *semiopen* interval includes the left boundary and excludes the right boundary:  $[l, r) = \{l, l + 1, \dots, r - 1\}$ .

Recall that the MERGESORT algorithm first sorts the left half of the given array, then sorts the second half, and finally merges the results.

The recursive implementation of this algorithm, given below, takes an array  $A$  as well as two indices  $l$  and  $r$  and sorts the subarray  $A[l..r]$ . That is, it sorts the *closed* interval  $[l, r] = \{l, l+1, \dots, r\}$  of  $A$  that includes both boundaries  $l$  and  $r$ .

```
MERGESORT( $A, l, r$ ):
if  $r - l + 1 \leq 1$ :
    return
 $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
MERGESORT( $A, l, m$ )
MERGESORT( $A, m + 1, r$ )
MERGE( $A, l, m, r$ )
```

Using *semiopen* instead of closed intervals reduces the chances of making an OBOB, because:

- (a) The number of elements in a semiopen interval  $[l, r)$  is  $r - l$  (for a closed interval  $[l, r]$ , it is  $r - l + 1$ ).
- (b) It is easy to split a semiopen interval into two semiopen intervals:  $[l, r) = [l, m) \cup [m, r)$  (for a closed interval  $[l, r]$ ,  $[l, r] = [l, m] \cup [m + 1, r]$ ).

Compare the previous implementation with the following one.

```
MERGESORT( $A, l, r$ ):
if  $r - l \leq 1$ :
    return
 $m \leftarrow l + \lfloor \frac{r-l}{2} \rfloor$ 
MERGESORT( $A, l, m$ )
MERGESORT( $A, m, r$ )
MERGE( $A, l, m, r$ )
```

For an array  $A[0..n - 1]$ , the outer call for the first implementation is  $\text{MERGESORT}(A, 0, n - 1)$ , while for the second one it is  $\text{MERGESORT}(A, 0, n)$ .

- LR3 Other things being equal, prefer row-by-row iteration through a matrix (i.e., a two-dimensional array) to a column-by-column iteration. The first approach is cache-friendly and hence usually runs faster.

## 8.2 C++ Specific

### 8.2.1 Code Format

CF1 Stick to a specific code style. Mixing various code styles in your programs make them less readable. Select your favorite code style and always follow it. Recommended style guides:

- C++ Core Guidelines by Bjarne Stroustrup and Herb Sutter: <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>
- Google C++ Style Guide: <http://google.github.io/styleguide/cppguide.html>

CF2 Be consistent. For example, either *always* put the opening curly bracket on the same line as the definition of a class or a function or *never* do it.

CF3 There are many ways of formatting C++ code and there is no universal way at the same time. It is important to be consistent (recall CF2). A simple and reasonable way of achieving this is autoformatting your code using clang-format.

### 8.2.2 Code Structure

CS1 Avoid statements of the form `using namespace foo`. This may potentially lead to name clashes (if, say, you implement your own `min` method) that will, in turn, lead to bugs that are difficult to catch. Moreover, this violates the general principle of using namespaces.

If you use `std::vector` and `std::cin` heavily in your program, instead of

```
using namespace std;
```

write

```
using std::vector;
using std::cin;
```

CS2 Avoid using old-style `scanf` and `printf`. Instead, use `std::cin` and `std::cout`.

For compatibility reasons, `iostream` synchronizes with `stdio` (which makes it possible to use both interfaces for input/output). Turning it off makes `std::cin` and `std::cout` work several times faster:

```
#include <iostream>

int main() {
    std::ios_base::sync_with_stdio(false);
    ...
    return 0;
}
```

This might be noticeable when, say, you output  $10^5$  integers.

CS3 Other things being equal, use pre-increment `++i` instead of post-increment `i++`. For integer types there is no difference, but for more complex iterators, the post-increment creates a copy of an object which, in turn, consumes more memory and slows down a program. At the same time, prefer range-based for loops when possible (see [LC5](#)).

CS4 Use `const` each time when it makes sense. This allows to catch some bugs at the compilation stage and serves as an additional documentation of your code.

CS5 Pass input parameters to a function by value in case they are of primitive types like `bool`, `double`, or `int32_t`. For more complex types of input parameters, pass them by a constant reference to avoid unnecessary copying.

CS6 When possible, use C++ constructions instead of the corresponding C constructions: e.g., `std::copy/std::copy_n` and `std::fill/std::fill_n` instead of `std::memcpy` and `std::memset`; `<limits>` instead of `<climits>`.

CS7 When using an external function or type, don't forget to include the corresponding header.

- CS8 Each variable should be used for a single purpose. Do not use auxiliary variable like `tmp` in various parts of your code. In most cases, variables containing `tmp` or `temp` in their names are either useless or named badly.
- CS9 Declare variables as close as possible to their first usage. This makes your code easier to read. Always initialize variables when declaring them (or immediately after declaration). Avoid using global variables and make each variable as local as possible (recall [LC4](#)): if it is used only in a function, it should be local for the function; if it is used only in a loop, it should be local for the loop. This allows to reduce the number of variables that should be kept in mind when reading your code.
- CS10 Use the `limits` header to check ranges.
- CS11 Do not use `std::pair`. This makes it difficult to read the code: each time a pair is used the reader has to go back to the definition of the pair to check what `first` and `second` mean. Moreover, it is particularly easy to confuse `first` and `second` in your code. Instead of a pair, define your own struct with appropriately named fields.  
The only exception of this rule is a case when a pair is used in a local piece of code that fits to one screen. In this case, it could be convenient to use a pair since it has a default comparing operator (that can be used, e.g., for sorting). The fact that it is only used in a short piece of code does not confuse the reader.
- CS12 Do not use `define` for declaring functions. Instead, use template or inline functions.
- CS13 C headers (with `.h` extension like `<stdio.h>`, `<math.h>`) are present in C++ due to backward compatibility only. Instead of them, use the corresponding C++ headers (like `<cstdio>`, `<cmath>`).

### 8.2.3 Types and Constants

- CT1 For Boolean type, use the `bool` type instead of `int`.
- CT2 Avoid using the following standard types: `short`, `long`, `long long`, `signed char`, `unsigned short`, `unsigned`, and `unsigned long long`. Prefer

`int32_t` and `int64_t` types (from the `cstdint` header) that are *guaranteed* to be 32 and 64 bit integers, respectively. Avoid using less than 32 bit types.

CT3 Do not subtract from an operand of unsigned type if you haven't verified that this will not lead to an overflow.

To illustrate this, assume that you need to iterate through all but the last five elements of container. The following code results in an undefined behavior in case container has less then five elements:

```
for (int i = 0; i < container.size() - 5; ++i) {  
    ...  
}
```

A safer way of doing this is the following:

```
for (std::size_t i = 0; i + 5 < container.size(); ++i) {  
    ...  
}
```

CT4 Avoid using `int[]` and `int*` for arrays. Use `std::vector` instead. Similarly, instead of C-type strings `char[]` and `char*`, use `std::string`.

CT5 Use C++ type cast (`static_cast<int>(x)`) instead of C (`((int)x)`) or functional style (`int(x)`) type cast (especially, in template code).

CT6 Do not use macros for defining constants. Instead of

```
#define MAX_LENGTH 100000
```

use either

```
constexpr int32_t kMaxLength = 100000;
```

or

```
const int32_t kMaxLength = 100000;
```

CT7 Use the following zero constants for various types: 0 for integer types, 0.0 for floating point types, nullptr for pointers, '\0' as null char.

CT8 Do not use magic constants in your code. Consider, e.g., the case when you need to iterate through all letters of an alphabet. Instead of

```
for (char letter = 'a'; letter <= 'z'; ++letter) {  
    ...  
}
```

use

```
const char FIRST = 'a';  
const char LAST = 'z';  
...  
for (char letter = FIRST; letter <= LAST; ++letter) {  
    ...  
}
```

#### 8.2.4 Classes

CC1 Differentiate class and struct (though the only formal difference between them is whether fields are public by default or not). Use struct when you need a bunch of *public* fields with no non-trivial methods. Use class as a bunch of *private* fields with methods processing them. Example of struct:

```
struct Point {  
    double x;  
    double y;  
};
```

```

bool operator < (const Point& first, const Point& second) {
    if (first.x != second.x) {
        return first.x < second.x;
    }
    return first.y < second.y;
}

```

Example of class:

```

class Path {
public:
    Path(double time, double average_speed)
        : time_(time), average_speed_(average_speed)
    {}

    double Time() const {
        return time_;
    }

    double AverageSpeed() const {
        return average_speed_;
    }

    double Distance() const {
        return time_ * average_speed_;
    }

private:
    double time_;
    double average_speed_;
};

```

CC2 When you need to compare objects of your class or struct, implement just operator<, but don't implement other operators for comparing (operator<=, operator>, operator>=). The reason is that the other

comparing operators are implied by operator<. Following this convention will decrease code duplication and make your program more readable.

If using standard C++2a, implement operator<=>.

CC3 When using member initializer lists in constructors, list the members in exactly the same order they are declared in the class definition. The reason is that the members are always initialized in order they are declared (rather than in the order they are listed). Hence, using different order may lead to difficult to catch bugs. E.g., the following code leads to undefined behavior.

```
struct DataHolder {
    explicit DataHolder(size_t size):
        data(size),
        current(&data[0])
    {}

    int* current;
    std::vector<int> data;
};
```

CC4 Use `explicit` specifier for constructors with one argument (to avoid implicit conversion and copy-initialization).

CC5 Use `final` and `override` specifiers whenever possible.

CC6 Use a separate naming style for private class members to make it easy to distinguish them from method parameters. One of the common conventions is to use either an underscore as a suffix (like `name_`) or the prefix `m_` (like `m_name`). Do not start the names with an underscore.

### 8.2.5 Containers

CCO1 Avoid using dynamically allocated variables explicitly (via `new` and `malloc`). This leads to memory leaks in case you forget to call `delete` or `free`. Moreover, `new` is quite slow, hence allocate small variables

on stack. For allocating data of size that is unknown in advance, use standard containers (like `std::vector`).

CCO2 When using standard containers, remember that they have various constructors, assignment operators, and other convenient methods. E.g., the following code creates an  $n \times m$  2d-array (i.e., a matrix) filled in by 100's.

```
std::vector< vector<int32_t> > cache(
    n,
    std::vector<int32_t>(m, 100));
```

The following code swaps two vectors without copying the whole contents:

```
std::vector<int32_t> first(1000000, 1);
std::vector<int32_t> second(2000000, 2);
first.swap(second);
```

## 8.2.6 Integers and Floating Point Numbers

CI1 Use `double` type for floating type numbers.

CI2 To compute the absolute value of a `double`, use `std::abs` from `cmath` header.

CI3 Do not compare `float` or `double` numbers using `<`, `>`, `<=`, `>=`, or `==`. The reason is that there is a precision loss when working with real numbers. Because of this, two sequences of operations that should lead to the same real number may result in different `float`/`double` numbers. A safer way of comparing `float`/`double` numbers is the following.

```
const double THRESHOLD = 1e-8;

bool isLess(double first, double second) {
    return first < second - THRESHOLD;
```

```

}

bool isLessOrEqual(double first, double second) {
    return first < second + THRESHOLD;
}

bool isEqual(double first, double second) {
    return std::abs(first - second) < THRESHOLD;
}

```

- CI4 For generating random numbers, use functions from the random header instead of the function `std::rand()`. Note that `std::rand()` generates integers not exceeding `RAND_MAX`. The value of `RAND_MAX` differs between different compilers.

## 8.3 Python Specific

### 8.3.1 General

- PG1 Follow PEP-8 style guide: <https://www.python.org/dev/peps/pep-0008/>. Use `pep8` (<https://pypi.python.org/pypi/pep8>) and `autopep8` (<https://pypi.python.org/pypi/autopep8>) tools.

- PG2 List imported modules in the beginning of a file. List them in the lexicographic order.

```

# bad
import gzip
import sys
from collections import defaultdict
import io
from contextlib import contextmanager
import functools
from urllib.request import urlopen

# good

```

```
import functools
import gzip
import io
import sys
from collections import defaultdict
from contextlib import contextmanager
from urllib.request import urlopen
```

PG3 Use operators `is` and `is_not` for comparing with singletons (like `None`) only. The only exception is Boolean constants `True` and `False`.

PG4 Use *falsy/truthy* semantics. *Falsy* values: `None`; `False`; zeroes `0`, `0.0`, and `0j`; empty strings and bytes; empty collections.

```
# bad
if acc == []:
    ...

# bad
if len(acc) > 0:
    ...

# good
if not acc:
    ...

# ok
if acc == 0:
    ...
```

PG5 Avoid unnecessary copying.

```
# bad
xs = set([x**2 for x in range(42)])

for x in list(sorted(xs)):
```

```
    ...
# good
xs = {x**2 for x in range(42)}

for x in sorted(xs):
    ...
```

PG6 Do not use the method `dict.get()`, do not use a collection `dict.keys` to check whether a key is present in `dict`, do not use `dict.keys` to iterate over a dictionary.

```
# bad
if key in dict.keys():
    ...

if not dict.get(key, False):
    ...

# good
if key in dict:
    ...

if key not in dict:
    ...

# bad
for key in dict.keys():
    ...

# good
for key in dict:
    ...
```

PG7 Use literals for creating empty collections. The only exception is set: there is no literal for an empty set in Python.

```
# bad
dict(), list(), tuple()

# good
{}, [], ()
```

PG8 In Python3, use // for integer division.

PG9 Use float("inf") for infinity.

PG10 If needed, increase the recursion depth as follows. (Note that to take advantage of bigger stack, one needs to launch the computation in a new thread.)

```
sys.setrecursionlimit(10 ** 7)
threading.stack_size(2 ** 27)
threading.Thread(target=main).start()
```

### 8.3.2 Code Structure

PC1 Do not emulate for loop.

```
# bad
i = 0
while i < n:
    ...
    i += 1

# good
for i in range(n):
    ...
```

PC2 Prefer range-based loops (see [LC5](#)). If an index is necessary, use enumerate.

```

# bad
for i in range(len(xs)) :
    x = xs[i]

# good
for x in xs:
    ...

# good
for i, x in enumerate(xs):
    ...

```

```

# bad
for i in range(min(len(xs), len(ys))) :
    f(xs[i], ys[i])

# good
for x, y in zip(xs, ys):
    f(x, y)

```

PC3 Do not use meaningless if and ternary operators (see [LC3](#)).

```

# bad
xs = [x for x in xs if predicate]
return True if xs else False

# good
xs = [x for x in xs if predicate]
return bool(xs)

# good
return any(map(predicate, xs))

```

PC4 Do not use `file.readline` and `file.readlines` for iterating over a file.

```
# bad
while True:
    line = file.readline()
    ...

for line in file.readlines():
    ...

# good
for line in file:
    ...
```

### 8.3.3 Functions

PF1 Avoid mutable default values.

PF2 Do not overuse functional idioms. In many cases, generators are more readable than a mix of `map`, `filter`, and `zip`.

```
# bad
list(map(lambda x: x ** 2,
         filter(lambda x: x % 2 == 1,
                range(10)))

# good
[x ** 2 for x in range(10) if x % 2 == 1]
```

PF3 Do not overuse generators. In many cases, a `for` loop is more readable.

PF4 Avoid meaningless anonymous functions.

```
# bad
map(lambda x: frobnicate(x), xs)

# good
map(frobnicate, xs)
```

```
# bad
collections.defaultdict(lambda: [])

# good
collections.defaultdict(list)
```

### 8.3.4 Strings

PS1 Use `str.startswith` and `str.endswith`.

```
# bad
s[:len(p)] == p
s.find(p) == len(s) - len(p)

# good
s.startswith(p)
s.endswith(p)
```

PS2 Use string formatting instead of concatenation of results of `str`.

```
# bad
"(+ " + str(expr1) + " " + str(expr2) + ")"

# good
"(+ {} {})".format(expr1, expr2)
```

Exception: converting a single parameter to a string.

```
# bad
"{}".format(value)

# good
str(value)
```

PS3 Simplify formatting expressions when possible.

```
# bad
"(+ {} {})".format(expr1, expr2)
"(+ {} {})".format(expr1, expr2)

# good
"(+ {} {})".format(expr1, expr2)
```

PS4 Keep in mind that `str.format` converts its parameters to strings.

```
# bad
"(+ {} {})".format(str(expr1), str(expr2))

# good
"(+ {} {})".format(expr1, expr2)
```

### 8.3.5 Classes

PCL1 Use `collections.namedtuple` as a collection of immutable fields.

```
# bad
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
# good
Point = namedtuple("Point", ["x", "y"])
```

PCL2 Avoid calling “magic methods” when there is an appropriate function or operator.

```
# bad
expr.__str__()
expr.__add__(other)

# good
str(expr)
expr + other
```

PCL3 Instead of using `type` for checking whether a given object has a required type, use the function `isinstance`.

```
# bad
type(instance) == Point
type(instance) is Point

# good
isinstance(instance, Point)
```

### 8.3.6 Exceptions

PE1 Keep the size of `try` and `with` blocks as small as possible.

PE2 Use `except Exception` rather than `except BaseException` or `except` for catching an exception.

PE3 Use the most specific type of exception in the `except` block.

```
# bad
try:
    mapping[key]
except Exception:
    ...

# good
try:
    mapping[key]
except KeyError:
    ...
```

PE4 Inherit your own exceptions from `Exception` rather than `BaseException`.

PE5 Use context managers instead of `try-finally`.

```
# bad
handle = open("path/to/file")
try:
    do_something(handle)
finally:
    handle.close()

# good
with open("path/to/file") as handle:
    do_something(handle)
```



# Appendix

## Compiler Flags

**C** (gcc 7.4.0). File extensions: .c. Flags:

```
gcc -pipe -O2 -std=c11 <filename> -lm
```

**C++** (g++ 7.4.0). File extensions: .cc, .cpp. Flags:

```
g++ -pipe -O2 -std=c++14 <filename> -lm
```

If your C/C++ compiler does not recognize `-std=c++14` flag, try replacing it with `-std=c++0x` flag or compiling without this flag at all (all starter solutions can be compiled without it). On Linux and macOS, you most probably have the required compiler. On Windows, you may use your favorite compiler or install, e.g., cygwin.

**C#** (mono 4.6.2). File extensions: .cs. Flags:

```
mcs
```

**Go** (golang 1.13.4). File extensions: .go. Flags

```
go
```

**Haskell** (ghc 8.0.2). File extensions: .hs. Flags:

```
ghc -O2
```

**Java** (OpenJDK 1.8.0\_232). File extensions: .java. Flags:

```
javac -encoding UTF-8  
java -Xmx1024m
```

**JavaScript** (NodeJS 12.14.0). File extensions: .js. No flags:

```
node js
```

**Kotlin** (Kotlin 1.3.50). File extensions: .kt. Flags:

```
kotlinc  
java -Xmx1024m
```

**Python** (CPython 3.6.9). File extensions: .py. No flags:

```
python3
```

**Ruby** (Ruby 2.5.1p57). File extensions: .rb.

```
ruby
```

**Rust** (Rust 1.37.0). File extensions: .rs.

```
rustc
```

**Scala** (Scala 2.12.10). File extensions: .scala.

```
scalac
```

## Frequently Asked Questions

### What Are the Possible Grading Outcomes?

There are only two outcomes: “pass” or “no pass.” To pass, your program must return a correct answer on all the test cases we prepared for you, and do so under the time and memory constraints specified in the problem statement. If your solution passes, you get the corresponding feedback

"Good job!" and get a point for the problem. Your solution fails if it either crashes, returns an incorrect answer, works for too long, or uses too much memory for some test case. The feedback will contain the index of the first test case on which your solution failed and the total number of test cases in the system. The tests for the problem are numbered from 1 to the total number of test cases for the problem, and the program is always tested on all the tests in the order from the first test to the test with the largest number.

Here are the possible outcomes:

- Good job! Hurrah! Your solution passed, and you get a point!
- Wrong answer. Your solution outputs incorrect answer for some test case. Check that you consider all the cases correctly, avoid integer overflow, output the required white spaces, output the floating point numbers with the required precision, don't output anything in addition to what you are asked to output in the output specification of the problem statement.
- Time limit exceeded. Your solution worked longer than the allowed time limit for some test case. Check again the running time of your implementation. Test your program locally on the test of maximum size specified in the problem statement and check how long it works. Check that your program doesn't wait for some input from the user which makes it to wait forever.
- Memory limit exceeded. Your solution used more than the allowed memory limit for some test case. Estimate the amount of memory that your program is going to use in the worst case and check that it does not exceed the memory limit. Check that your data structures fit into the memory limit. Check that you don't create large arrays or lists or vectors consisting of empty arrays or empty strings, since those in some cases still eat up memory. Test your program locally on the tests of maximum size specified in the problem statement and look at its memory consumption in the system.
- Cannot check answer. Perhaps the output format is wrong. This happens when you output something different than expected. For example, when you are required to output either "Yes" or "No",

but instead output 1 or 0. Or your program has empty output. Or your program outputs not only the correct answer, but also some additional information (please follow the exact output format specified in the problem statement). Maybe your program doesn't output anything, because it crashes.

- **Unknown signal 6** (or 7, or 8, or 11, or some other). This happens when your program crashes. It can be because of a division by zero, accessing memory outside of the array bounds, using uninitialized variables, overly deep recursion that triggers a stack overflow, sorting with a contradictory comparator, removing elements from an empty data structure, trying to allocate too much memory, and many other reasons. Look at your code and think about all those possibilities. Make sure that you use the same compiler and the same compiler flags as we do.
- **Internal error:** exception... Most probably, you submitted a compiled program instead of a source code.
- **Grading failed.** Something wrong happened with the system. Report this through Coursera or edX Help Center.

## Why the Test Cases Are Hidden?

See section [3.2.4](#).

## May I Post My Solution at the Forum?

Please do not post any solutions at the forum or anywhere on the web, even if a solution does not pass the tests (as in this case you are still revealing parts of a correct solution). Our students follow the Honor Code: “I will not make solutions to homework, quizzes, exams, projects, and other assignments available to anyone else (except to the extent an assignment explicitly permits sharing solutions).”

## Do I Learn by Trying to Fix My Solution?

*My implementation always fails in the grader, though I already tested and stress tested it a lot. Would not it be better if you gave me a solution to this problem*

*or at least the test cases that you use? I will then be able to fix my code and will learn how to avoid making mistakes. Otherwise, I do not feel that I learn anything from solving this problem. I am just stuck.*

First of all, learning from your mistakes is one of the best ways to learn.

The process of trying to invent new test cases that might fail your program is difficult but is often enlightening. Thinking about properties of your program makes you understand what happens inside your program and in the general algorithm you're studying much more.

Also, it is important to be able to find a bug in your implementation without knowing a test case and without having a reference solution, just like in real life. Assume that you designed an application and an annoyed user reports that it crashed. Most probably, the user will not tell you the exact sequence of operations that led to a crash. Moreover, there will be no reference application. Hence, it is important to learn how to find a bug in your implementation yourself, without a magic oracle giving you either a test case that your program fails or a reference solution. We encourage you to use programming assignments in this class as a way of practicing this important skill.

If you have already tested your program on all corner cases you can imagine, constructed a set of manual test cases, applied stress testing, etc, but your program still fails, try to ask for help on the forum. We encourage you to do this by first explaining what kind of corner cases you have already considered (it may happen that by writing such a post you will realize that you missed some corner cases!), and only afterward asking other learners to give you more ideas for tests cases.