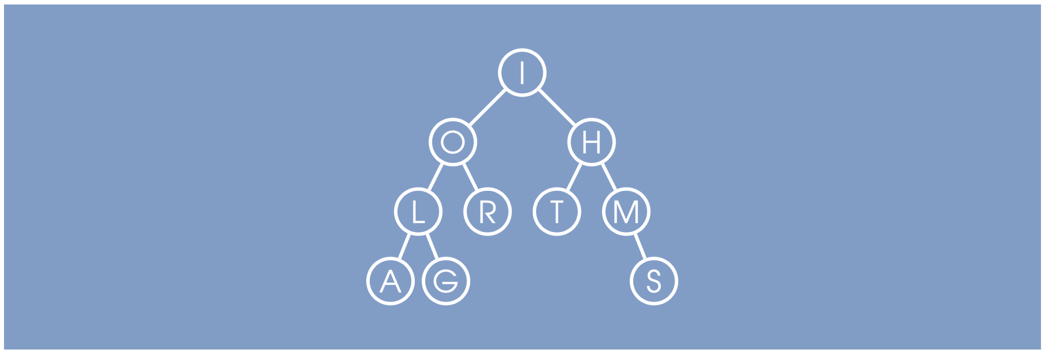


Welcome to the Algorithmic Toolbox course at Coursera! This file contains the statements of 32 programming challenges.

Practice writing efficient and reliable code. The programming challenges represent the most important feature of this course: we believe that implementing an algorithm is a crucial computer science skill. You can learn more about our teaching philosophy here. You will be practicing writing efficient and reliable code for a multitude of algorithmic problems using any of 12 programming languages: `C++`, `Java`, `Python`, `C`, `C#`, `Go`, `Haskell`, `JavaScript`, `Kotlin`, `Ruby`, `Rust`, `Scala`.

Additional useful resources. You may download all slides and video recordings from this course here.

Prepare for you next coding interview. Many students use our "Algorithmic Toolbox" MOOC to prepare for their next coding interview. Our interactive textbook "Ace Your Next Coding Interview by Learning Algorithms through Programming and Puzzle Solving" prepares you for a coding interview by covering all programming challenges and puzzles in "Algorithmic Toolbox". It further extends them by describing many challenges that you are likely to encounter on your next interview. The book also discusses good programming practices that will help you to become a better programmer and illustrates their use by providing `Python` solutions for all problems in the "Algorithmic Toolbox". It also has a Teaching Assistant who responds to your comments and even holds office hours on Zoom to answer whatever questions you may have about algorithms.

Contents

1 Programming Challenges **7**
1.1 Sum of Two Digits 8
1.1.1 Solutions in Various Programming Languages 9
1.1.2 Submitting to the Grading System at Coursera . . . 11
1.2 Maximum Pairwise Product 12
1.2.1 Naive Algorithm 13
1.2.2 Fast Algorithm 17
1.2.3 Testing and Debugging 18
1.2.4 Can You Tell Me What Error Have I Made? 20
1.2.5 Stress Testing 20
1.2.6 Even Faster Algorithm 25
1.2.7 A More Compact Algorithm 25
1.3 Solving a Programming Challenge in Five Easy Steps 25
1.3.1 Reading Problem Statement 26
1.3.2 Designing an Algorithm 26
1.3.3 Implementing an Algorithm 26
1.3.4 Testing and Debugging 27
1.3.5 Submitting to the Grading System 28

2 Algorithmic Warm Up **29**
2.1 Programming Challenges 31
2.1.1 Fibonacci Number 31
2.1.2 Last Digit of Fibonacci Number 33
2.1.3 Huge Fibonacci Number 35
2.1.4 Last Digit of the Sum of Fibonacci Numbers 37
2.1.5 Last Digit of the Partial Sum of Fibonacci Numbers . 38
2.1.6 Last Digit of the Sum of Squares of Fibonacci Numbers 39
2.1.7 Greatest Common Divisor 41
2.1.8 Least Common Multiple 42
2.2 Summary of Algorithmic Ideas 43

3 Greedy Algorithms **45**
3.1 The Main Idea 48
3.1.1 Examples 48

3.1.2 Proving Correctness of Greedy Algorithms 49
3.1.3 Implementation 50
3.2 Programming Challenges 51
3.2.1 Money Change 51
3.2.2 Maximum Value of the Loot 52
3.2.3 Car Fueling 54
3.2.4 Maximum Advertisement Revenue 56
3.2.5 Collecting Signatures 58
3.2.6 Maximum Number of Prizes 60
3.2.7 Maximum Salary 62

4 Divide-and-Conquer **65**
4.1 The Main Idea 67
4.1.1 Guess a Number 67
4.1.2 Searching Sorted Data 71
4.1.3 Finding a White-Black Pair 73
4.1.4 Finding a Peak 75
4.1.5 Multiplying Integers 76
4.1.6 The Master Theorem 80
4.2 Programming Challenges 87
4.2.1 Binary Search 87
4.2.2 Binary Search with Duplicates 90
4.2.3 Majority Element 92
4.2.4 Speeding-up RandomizedQuickSort 93
4.2.5 Number of Inversions 95
4.2.6 Organizing a Lottery 97
4.2.7 Closest Points 99

5 Dynamic Programming **103**
5.1 The Main Idea 106
5.1.1 Number of Paths 106
5.1.2 Dynamic Programming 108
5.1.3 Shortest Path in Directed Acyclic Graph 109
5.2 Programming Challenges 111
5.2.1 Money Change Again 112
5.2.2 Primitive Calculator 113
5.2.3 Edit Distance 115
5.2.4 Longest Common Subsequence of Two Sequences . . 117

5.2.5 Longest Common Subsequence of Three Sequences . 119
5.2.6 Maximum Amount of Gold 121
5.2.7 Splitting the Pirate Loot 123
5.2.8 Maximum Value of an Arithmetic Expression 125
5.3 Designing Dynamic Programming Algorithms 126

Appendix **127**
Compiler Flags 127
Frequently Asked Questions 128

Chapter 1: 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.

1.1 Sum of Two Digits

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

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

$$2+3=5$$

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

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

Output format. The sum of a and b.

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

Sample.

Input:

```
9 7
```

Output:

```
16
```

Time limits (sec.):

C++	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.

1.1.1 Solutions in Various Programming Languages

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++

```cpp
#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

```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

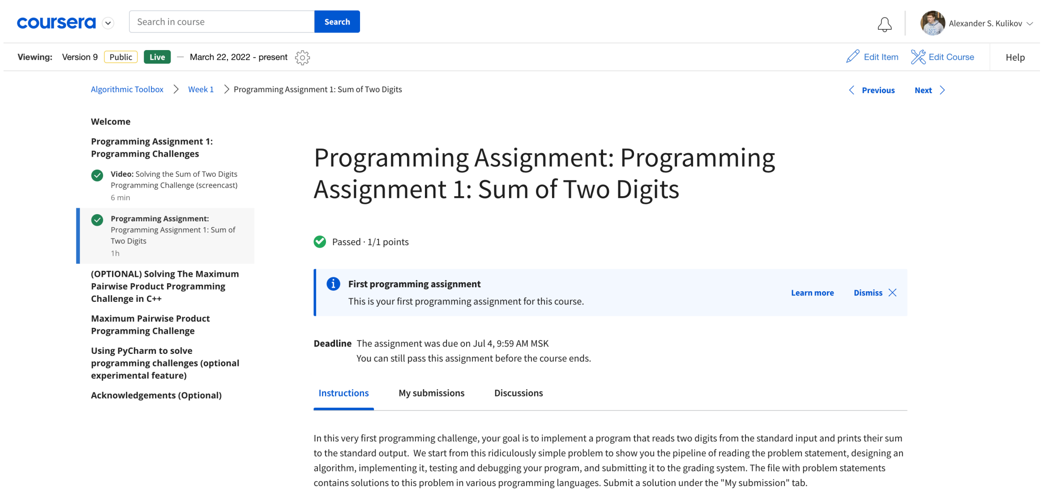
```python
def sum_of_digits(first_digit, second_digit):
    return first_digit + second_digit

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).

1.1.2 Submitting to the Grading System at Coursera

This is what a fresh submission page looks like:


coursera
Search in course
Search
Alexander S. Kulikov
Viewing: Version 9 Public Live — March 22, 2022 - present
Edit Item
Edit Course
Help
Algorithmic Toolbox > Week 1 > Programming Assignment 1: Sum of Two Digits
Previous
Next
Welcome
Programming Assignment 1: Programming Challenges
Video: Solving the Sum of Two Digits Programming Challenge (screencast) 6 min
Programming Assignment: Programming Assignment 1: Sum of Two Digits 1h
(OPTIONAL) Solving The Maximum Pairwise Product Programming Challenge in C++
Maximum Pairwise Product Programming Challenge
Using PyCharm to solve programming challenges (optional experimental feature)
Acknowledgements (Optional)
Learn more
Dismiss
Instructions
My submissions
Discussions

Programming Assignment: Programming Assignment 1: Sum of Two Digits

Passed · 1/1 points

First programming assignment
This is your first programming assignment for this course.

Deadline The assignment was due on Jul 4, 9:59 AM MSK
You can still pass this assignment before the course ends.

In this very first programming challenge, your goal is to implement a program that reads two digits from the standard input and prints their sum to the standard output. 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. The file with problem statements contains solutions to this problem in various programming languages. Submit a solution under the "My submission" tab.

If this is you first programming assignment at Coursera, we encourage you to go through general information about programming assignments ("Learn more" button in the blue bubble).

The line below the blue bubble indicates the deadline for this assignment. Please not that this is a suggestion rather than a hard deadline. If you miss a deadline, you may safely switch to the next session of the course (a new session starts automatically every two weeks). All your progress will be transferred to the new session in this case. See Coursera help article on switching sessions.

The "Discussion" tab leads to a forum attached to this assignment. Please post your questions and help other learners there.

To submit your solution, go to the "My submissions" tab, press the "Create submission" button, upload the source code of your solution, and press the "Submit" button. You may safely leave the page while your solution is being graded (in most cases, it is done in less than a minute, but when the servers are overloaded it may take several minutes; please be patient).

1.2 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 $a_1,\ldots,a_n$, compute

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

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

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

Output format. The maximum pairwise product.

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

Sample 1.

Input:

```
3
1 2 3
```

Output:

```
6
```

Sample 2.

Input:

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

Output:

```
140
```

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

1.2.1 Naive Algorithm

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

```
MaxPairwiseProductNaive($A[1\ldots n]$):
$product \leftarrow 0$
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($A[1\ldots n]$):
$product \leftarrow 0$
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!

1.2.2 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 $A[index_1] \cdot A[index_2]$
```

1.2.3 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 410 065 408 or even a negative number instead of the correct result 9 000 000 000. If it does, this is most probably caused by an *integer overflow*. For example, in C++ programming language a large number like 9 000 000 000 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 $1 \le i \le 2\cdot 10^5$. There are two ways of doing this.

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

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

   ```
   yourprogram < dataset.txt
   ```

Check that your program processes this dataset within time limit and returns the correct result: 39 999 800 000. 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?

1.2.4 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.

1.2.5 Stress Testing

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

A stress test consists of four parts:

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

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

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

```
StressTest($N, M$):
while true:
  $n \leftarrow$ random integer between 2 and $N$
  allocate array $A[1 \ldots n]$
  for $i$ from 1 to $n$:
    $A[i] \leftarrow$ random integer between 0 and $M$
  print($A[1 \ldots n]$)
  $result_1 \leftarrow$ 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 StressTest(10, 100 000) and keep our fingers crossed in a hope that it outputs "Wrong answer." We see something like this (the result can be different on your computer because of a different random number generator).

```
...
OK
67232 68874 69499
OK
6132 56210 45236 95361 68380 16906 80495 95298
OK
62180 1856 89047 14251 8362 34171 93584 87362 83341 8784
OK
21468 16859 82178 70496 82939 44491
OK
68165 87637 74297 2904 32873 86010 87637 66131 82858 82935
Wrong answer: 7680243769 7537658370
```

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

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

Note that generating tests automatically and running stress test is easy, but debugging is hard. Before diving into debugging, let's try to generate a smaller test case to simplify it. To do that, we change N from 10 to 5 and M from 100 000 to 9.

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

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

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

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

Stop and Think. How MaxPairwiseProductFast can possibly return 27?

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

```
print(index_1, index_2)
```

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

```
...
7 3 6
1 3
OK
5
2 9 3 1 9
2 3
Wrong answer: 81 27
```

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

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

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

to

```
$i \neq index_1$
```

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

However, you shouldn't stop here, since you have only generated very small tests with $N = 5$ and $M = 10$. We should check whether our program works for larger n and larger elements of the array. So, we change N to 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($A[1 \ldots n]$):
$index \leftarrow 1$
for $i$ from 2 to $n$:
    if $A[i] > A[index]$:
        $index \leftarrow i$
swap $A[index]$ and $A[n]$
$index \leftarrow 1$
for $i$ from 2 to $n - 1$:
    if $A[i] > A[index]$:
        $index \leftarrow i$
swap $A[index]$ and $A[n - 1]$
return $A[n - 1] \cdot A[n]$
```

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.

1.2.6 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?

1.2.7 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($A[1 \ldots n]$):
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 ($2 \le n \le 2 \cdot 10^5$) this is usually sufficiently fast to fit into a second and pass our grader.

1.3 Solving a Programming Challenge in Five Easy Steps

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

1.3.1 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.

1.3.2 Designing an Algorithm

After designing an algorithm, prove that it is correct and try to estimate its expected running time on the most complex inputs specified in the constraints section. If you laptop performs roughly 10^8–10^9 operations per second, and the maximum size of a dataset in the problem description is $n = 10^5$, then an algorithm with quadratic running time is unlikely to fit into the time limit (since $n^2 = 10^{10}$), while a solution with running time $O(n \log n)$ will. However, an $O(n^2)$ solution will fit if $n = 1\,000$, and if $n = 100$, even an $O(n^3)$ solutions will fit. Although polynomial algorithms remain unknown for some hard problems in this book, a solution with $O(2^n n^2)$ running time will probably fit into the time limit as long as n is smaller than 20.

1.3.3 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).

1.3.4 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 $0, 0, \ldots, 0$ or a sequence $10^6, 10^6, \ldots, 10^6$. 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.

1.3.5 Submitting to the Grading System

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

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

Chapter 2: 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.

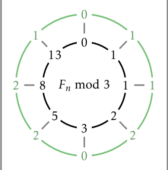
You may think that we are obsessed with the Fibonacci numbers because the first six challenges in this section represent questions about them. However, we start from seemingly similar Fibonacci problems because they gradually become more and more difficult — each new problem requires a development of 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 constrains, 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!

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

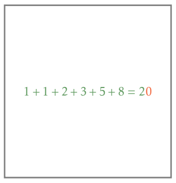
Fibonacci Number
Compute the n-th Fibonacci number.
Input. An integer n.
Output. n-th Fibonacci number.

$F_{100} = 354224848179261915075$

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.

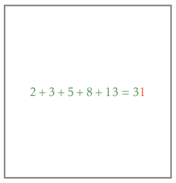


Huge Fibonacci Number
Compute the n-th Fibonacci number modulo m.
Input. Integers n and m.
Output. n-th Fibonacci number modulo m.



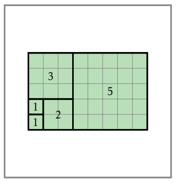
Last Digit of the Sum of Fibonacci Numbers

Compute the last digit of $F_0 + F_1 + \cdots + F_n$.
Input. An integer n.
Output. The last digit of $F_0 + F_1 + \cdots + F_n$.



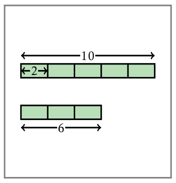
Last Digit of the Partial Sum of Fibonacci Numbers

Compute the last digit of $F_m + F_{m+1} + \cdots + F_n$.
Input. Integers $m \leq n$.
Output. The last digit of $F_m + F_{m+1} + \cdots + F_n$.



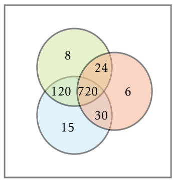
Last Digit of the Sum of Squares of Fibonacci Numbers

Compute the last digit of $F_0^2 + F_1^2 + \cdots + F_n^2$.
Input. An integer n.
Output. The last digit of $F_0^2 + F_1^2 + \cdots + F_n^2$.



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.

2.1 Programming Challenges

2.1.1 Fibonacci Number

Fibonacci Number Problem
Compute the n-th Fibonacci number.

Input: An integer n.
Output: n-th Fibonacci number.

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

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 $n \leq 1$:
    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 1.3.1.

2.1.2 Last Digit of Fibonacci Number

Last Digit of Fibonacci Number Problem
Compute the last digit of the n-th Fibonacci number.

> **Input:** An integer n.
> **Output:** The last digit of the n-th Fibonacci number.

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

Input format. An integer n.

Output format. The last digit of F_n.

Constraints. $0 \le n \le 10^6$.

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:

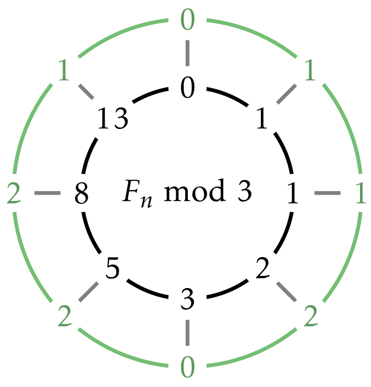
```
6
```

$F_{91\,239}$ will take more than ten pages to write down, but its last digit is 6.

2.1.3 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 mod m.

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

Sample 1.

Input:

```
1 239
```

Output:

```
1
```

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

Sample 2.

Input:

```
115 1000
```

Output:

```
885
```

$F_{115} \bmod 1000 = 483\,162\,952\,612\,010\,163\,284\,885 \bmod 1000 = 885$.

Sample 3.

Input:

```
2816213588 239
```

Output:

```
151
```

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

2.1.4 Last Digit of the Sum of Fibonacci Numbers

Last Digit of the Sum of Fibonacci Numbers Problem
Compute the last digit of $F_0 + F_1 + \cdots + F_n$.

> **Input:** An integer n.
> **Output:** The last digit of $F_0 + F_1 + \cdots + F_n$.

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

Input format. An integer n.

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

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
```

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

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

2.1.5 Last Digit of the Partial Sum of Fibonacci Numbers

Last Digit of the Partial Sum of Fibonacci Numbers Problem
Compute the last digit of $F_m + F_{m+1} + \cdots + F_n$.

> **Input:** Integers $m \le n$.
> **Output:** The last digit of $F_m + F_{m+1} + \cdots + F_n$.

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

Input format. Integers m and n.

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

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

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:

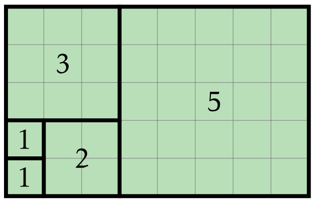
```
5
```

$F_{10} = 55$.

2.1.6 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 $F_0^2 + F_1^2 + \cdots + F_n^2$.

> **Input:** An integer n.
> **Output:** The last digit of $F_0^2 + F_1^2 + \cdots + F_n^2$.



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 $F_0^2 + F_1^2 + \cdots + F_n^2$. 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. $F_0^2 + F_1^2 + \cdots + F_n^2 \bmod 10$.

Constraints. $0 \le n \le 10^{14}$.

Sample 1.

Input:

```
7
```

Output:

```
3
```

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

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
```

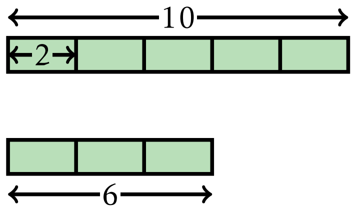
Output:

```
0
```

2.1.7 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 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. GCD(a,b).

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

Sample.

Input:

```
28851538 1183019
```

Output:

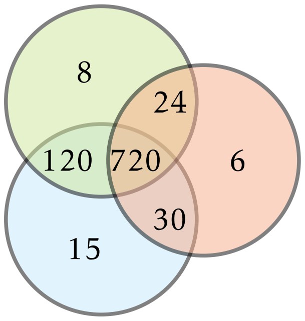
```
17657
```

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

2.1.8 Least Common Multiple

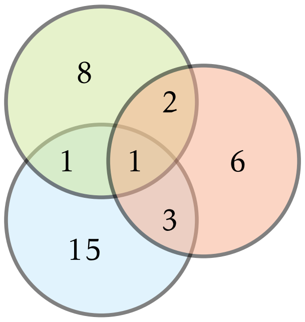
Least Common Multiple Problem
Compute the least common multiple of two positive integers.

> **Input:** Two positive integers a and b.
> **Output:** The least common multiple of a and b.



The least common multiple $\mathrm{LCM}(a,b)$ 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 $\mathrm{LCM}(a,b)$ is related to $\mathrm{GCD}(a,b)$?

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

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

Constraints. $1 \le a,b \le 10^7$.

Sample.

Input:

```
761457 614573
```

Output:

```
467970912861
```

2.2 Summary of Algorithmic Ideas

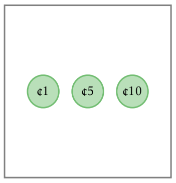
Now when you've implemented several algorithms, let's summarize and review the main ideas that we've discussed in this chapter.

What solution fits into a second? Modern computers perform about 10^8–10^9 basic operations per second. If you 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 3: 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

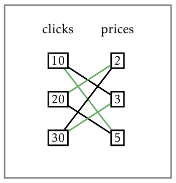
Find the maximal value of items that fit into the backpack.
Input. The capacity of a backpack W as well as the weights $(w_1,\ldots,w_n)$ and costs $(c_1,\ldots,c_n)$ of n different compounds.
Output. The maximum total value of fractions of items that fit into the backpack of the given capacity: i.e., the maximum value of $c_1 \cdot f_1 + \cdots + c_n \cdot f_n$ such that $w_1 \cdot f_1 + \cdots + w_n \cdot f_n \leq W$ 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 $stop_1 < stop_2 < \cdots < stop_n$.

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 $stop_1, stop_2, \ldots, stop_n$ along the way. We assume that a car starts with a full tank.

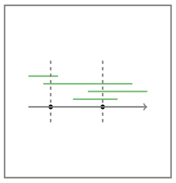


Maximum Product of Two Sequences

Find the maximum dot product of two sequences of numbers.

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

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

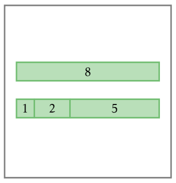


Covering Segments by Points

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

Input. A sequence of n segments $[l_1, r_1], \ldots, [l_n, r_n]$ 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 $l_i \le x \le r_i$.



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 $a_1 + \cdots + a_k$ 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.

3.1 The Main Idea

3.1.1 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 ← empty string
while Numbers is not empty:
   maxNumber ← 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 ← 0
while money > 0:
   maxCoin ← largest among Denominations that does not exceed money
   money ← money − maxCoin
   numCoins ← 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 $8 = 4 + 4$!

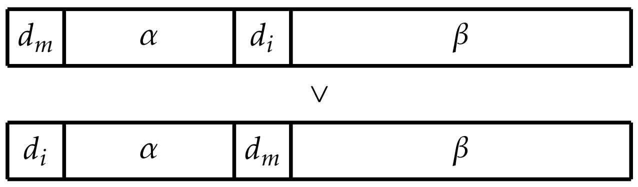
Thus, in *rare* cases when a greedy strategy works, one should be able to prove its correctness: A priori, there should be no reason why a sequence of *locally* optimal moves leads to a *global* optimum!

3.1.2 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 3.2.1). Let N be the largest number that can be obtained by concatenating digits $d_1, \ldots, d_n$ 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 α, β 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?

3.1.3 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.

3.2 Programming Challenges

3.2.1 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. $1 \le money \le 10^3$.

Sample 1.

Input:

```
2
```

Output:

```
2
```

$2 = 1 + 1$.

Sample 2.

Input:

```
28
```

Output:

```
6
```

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

3.2.2 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 $(w_1, \ldots, w_n)$ and costs $(c_1, \ldots, c_n)$ 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 $c_1 \cdot f_1 + \cdots + c_n \cdot f_n$ such that $w_1 \cdot f_1 + \cdots + w_n \cdot f_n \le W$ and $0 \le f_i \le 1$ 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

$$5\,000 \cdot \frac{u_1}{4} + 200 \cdot \frac{u_2}{3} + 10 \cdot \frac{u_3}{5}.$$

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

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. $1 \le n \le 10^3$, $0 \le W \le 2 \cdot 10^6$; $0 \le c_i \le 2 \cdot 10^6$, $0 < w_i \le 2 \cdot 10^6$ for all $1 \le i \le n$. 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.

3.2.3 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 $stop_1 < stop_2 < \cdots < stop_n$.
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 $stop_1, stop_2, \ldots, stop_n$ 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 $stop_1, stop_2, \ldots, stop_n$.

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

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

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:

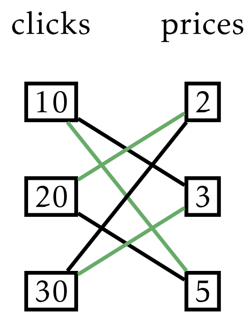
```
0
```

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.

3.2.4 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: $price_1, \ldots, price_n$ and $clicks_1, \ldots, clicks_n$.
Output: The maximum value of $price_1 \cdot c_1 + \cdots + price_n \cdot c_n$, where $c_1, \ldots, c_n$ is a permutation of $clicks_1, \ldots, clicks_n$.



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 $10\cdot5+20\cdot2+30\cdot3 = 180$ dollars, while the black one results in revenue of $10\cdot3+20\cdot5+30\cdot2 = 190$ dollars.

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

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

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

Sample 1.

Input:

```
1
23
39
```

Output:

```
897
```

$897 = 23 \cdot 39$.

Sample 2.

Input:

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

Output:

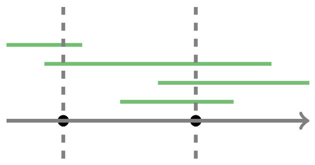
```
79
```

$79 = 7 \cdot 9 + 2 \cdot 2 + 3 \cdot 4$.

3.2.5 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 $[l_1, r_1], \ldots, [l_n, r_n]$ 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 $l_i \le x \le r_i$.



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. $1 \le n \le 100$; $0 \le l_i \le r_i \le 10^9$ 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:

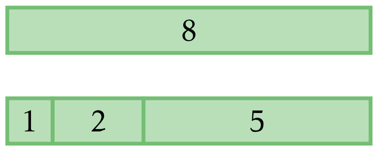
```
2
3 6
```

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.

3.2.6 Maximum Number of Prizes

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

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



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. $1 \le n \le 10^9$.

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
```

3.2.7 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.

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 ≥ 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 $number \geq maxNumber$:
```

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
```

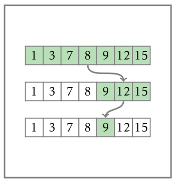
Output:

```
923923
```

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

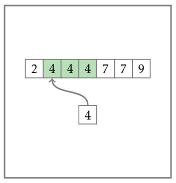
Chapter 4: Divide-and-Conquer

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 $Q = \{q_0, \ldots, q_{m-1}\}$ of integers.
Output. For each q_i, check whether it occurs in K.

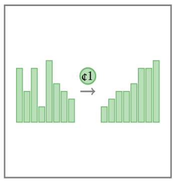


Binary Search with Duplicates

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.

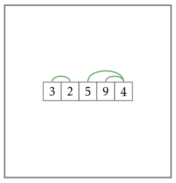


Speeding-up RandomizedQuickSort

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.

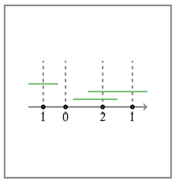


Number of Inversions

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 $a_i > a_j$.

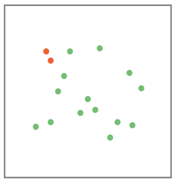


Points and Segments

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.

4.1 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.

4.1.1 Guess a Number

Interactive Puzzle "Guess a Number". In the "Guess a Number" game, your opponent has an integer $1 \le x \le n$ 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 $1 \le x \le 3$ 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 $1 \le x \le 7$ 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 4.1.

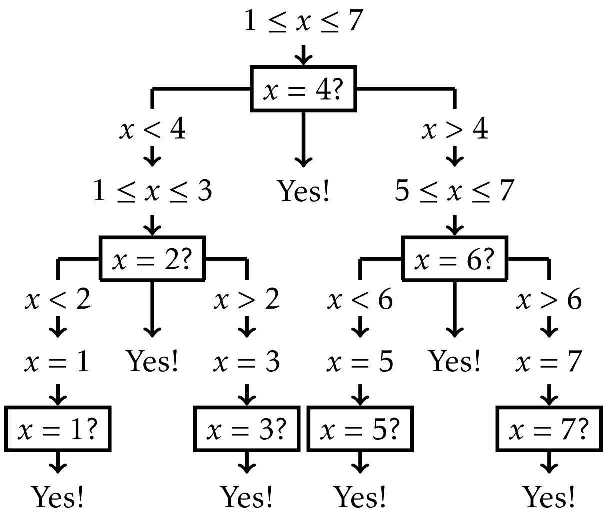


Figure 4.1: Guessing an integer $1 \le x \le 7$ by asking at most three questions.

This strategy allows you to guess an integer $1 \le x \le 2\,097\,151$ (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

$$\texttt{lower} \le x \le \texttt{upper},$$

that is, x lies in the segment [`lower`, `upper`]. It first computes the `middle` point of the segment [`lower`, `upper`] and then calls `query(middle)`. If $x <$ `middle`, then it continues with the interval [`lower`, `middle - 1`]. If $x >$ `middle`, then it continues with the interval [`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 $\log_2 n < 30$.

```python
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
```

4.1.2 Searching Sorted Data

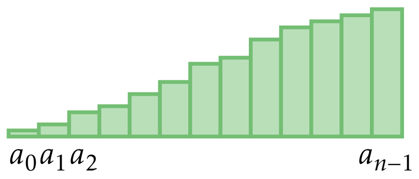
The method that we used for guessing a number is known as the *binary search*. Perhaps the most important application of binary search is *searching sorted data*. Searching is a fundamental problem: given a sequence and an element x, we would like to check whether x is present in this sequence. For example, 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
```

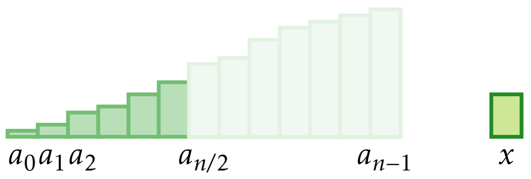
What is going on under the hood when one calls this `in` method? As you would expect, Python simply performs a *linear scan*. This linear scan makes up to n comparisons on a sequence of length n. If the sequence does not contain x, we *have to* scan all the elements: if we skip an element, we can't be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

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

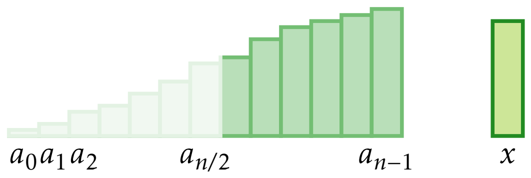


It turns out that in this case about $\log_2 n$ comparisons are enough! This is a great speedup: the linear scan of a sorted array with a billion elements will take a billion comparisons, but binary search makes at most $\log_2 10^9 < 30$ comparisons!

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:



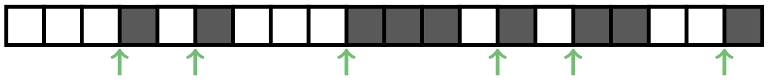
This leads us to the following code.

```
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)
```

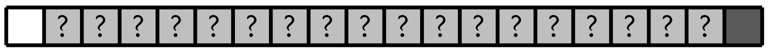
```
Searching 8 in [1, 2, 3, 3, 5, 6, 6, 8, 9, 9, 9]
Searching 8 in [6, 8, 9, 9, 9]
Searching 8 in [6, 8]
Found!
```

4.1.3 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:

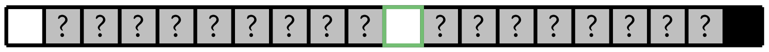


You may point to any cell and ask a question "What is its color?" Your goal is to find a white-black pair by asking the minimum number of questions. Try it online! After solving this puzzle, you will see that the binary search is useful even when the data is not sorted!

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.

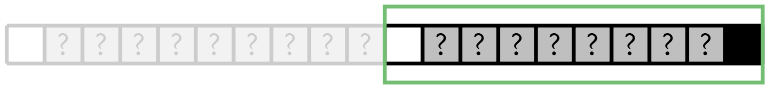
Even though this proof works for any array that starts from a white cell and ends in a black cell, it is *non-constructive*: it proves that there exists a white-black pair, but doesn't give an *efficient method* for finding this pair (by revealing the color of a few cells). In particular, if we start 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.



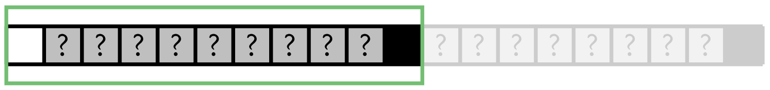
Stop and Think. If the middle cell turned out to be white, how would you proceed?

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).



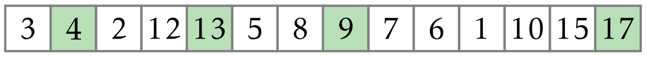
Hence, in any case, we decrease the size by a factor of (almost) two: from a starting array of length n, we get an array of length $\lceil\frac{n+1}{2}\rceil$. Thus, after revealing the colors of at most five cells, we will find a white-black pair. Indeed, even in the worst possible case, the size of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

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

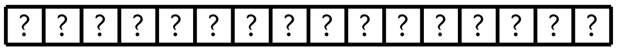
When the size of the current sequence is reduced to two, we are left with an array consisting of just two cells, these two cells form a white-black since the first of them is white and the last is black.

4.1.4 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.



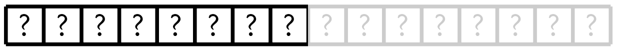
Interactive Puzzle "Finding a Peak". Consider an integer sequence with distinct unknown integers.



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

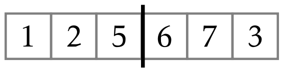
Note that any sequence (consisting of distinct integers) contains a peak: for example, the largest element is a peak. The largest element is a *global maximum* and it is not difficult to show that one needs to read the entire sequence to find it: if some element is not revealed, one cannot be sure that 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.

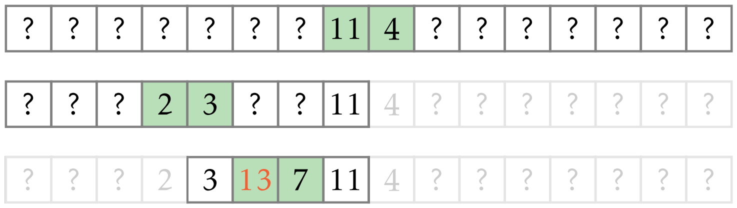
Indeed, as the example below illustrates, a peak in the left half, if it represents the last element in this half, is not necessarily a peak in the entire sequence.



Below, we denote the last element in the left half as *left* and the first element in the right half as *right*. If $\mathit{left} > \mathit{right}$, 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 $\mathit{left} < \mathit{right}$? 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.



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 $2\log_2 n$.

4.1.5 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:

$$\begin{array}{cccccc}
 & & & & 2 & 5 \\
 & & & \times & 6 & 3 \\
\hline
 & & & & 1 & 5 \\
 & & & & 6 & 0 \\
+ & & & 3 & 0 & 0 \\
 & & 1 & 2 & 0 & 0 \\
\hline
 & & 1 & 5 & 7 & 5
\end{array}$$

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?

$$(10A+B)\times(10C+D) = {\color{blue}100(A\times C)}+{\color{red}10(A\times D)+10(B\times C)}+{\color{green}B\times D}\,.$$

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:

$$(A+B)\times(C+D) - A\times C - B\times D = A\times D + B\times C\,. \tag{4.1}$$

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 $X = 54\,192\,143$ and $Y = 885\,274$, we can split X into parts $A = 5\,419$ and $B = 2\,143$ and split Y into parts $C = 0\,088$ and $D = 5\,274$ (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,

$$X = 54\,192\,143 = 54\,190\,000 + 2\,143 = 10^4A + B\,,$$
$$Y = 885\,274 = 880\,000 + 5\,274 = 10^4C + D\,.$$

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 there 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 (4.1) 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 $n \le 1$:
  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

$$T(n) \le 3T(n/2) + O(n).$$

Indeed, it makes three recursive calls for $n/2$-digit integers.[1] 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

$$T(n) \le O(n^{\log_2 3}) = O(n^{1.584\ldots}).$$

[1]We are cheating a bit: the integers $A + B$ and $C + D$ may have $n/2 + 1$ digits. Still, the recurrence $T(n) \le 3T(n/2 + 1) + O(n)$ implies the same bound.

4.1.6 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.[2] 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,

$$T(n) = T\left(\frac{n}{2}\right) + O(1).$$

[2]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 $(1, \alpha, \alpha^2, \ldots, \alpha^n)$ is called a *geometric progression* with ratio α. Its sum

$$\sum_{i=0}^{n} \alpha^i = 1 + \alpha + \cdots + \alpha^n$$

is known as the *geometric series*.

Lemma 4.1.1 (order of growth of the geometric series)**.** *Let α be a positive constant. Then,*

$$\sum_{i=0}^{n} \alpha^i = \begin{cases} \Theta(\alpha^n) & \text{if } \alpha > 1, \\ \Theta(n) & \text{if } \alpha = 1, \\ \Theta(1) & \text{if } \alpha < 1. \end{cases} \tag{4.2}$$

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:

$$\sum_{i=0}^{n} \alpha^i = \frac{\alpha^{n+1}-1}{\alpha-1} \tag{4.3}$$

Now, consider three cases.

1. $\alpha > 1$. Then, $\alpha^n(\alpha-1) < \alpha^{n+1}-1 < \alpha^{n+1}$ and (4.3) 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 (4.3), but the series is equal to $n+1$.

3. $\alpha < 1$. Then, $1-\alpha < 1-\alpha^{n+1} < 1$ and (4.3) 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.

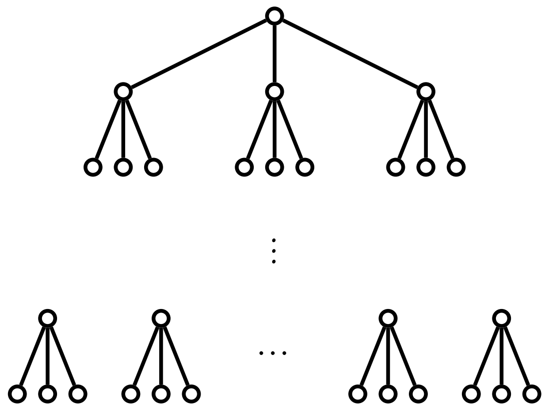
$$\begin{aligned}
T(n) &= cn^d + aT(n/b)\\
&= cn^d + a(c(n/b)^d + aT(n/b^2)) = cn^d + ca(n/b)^d + a^2T(n/b^2) =\\
&\vdots\\
&= cn^d + ca(n/b)^d + ca^2(n/b^2)^d + \cdots =\\
&= \sum_{l=0}^{\log_b n} ca^l\left(\frac{n}{b^l}\right)^d = \qquad (4.4)\\
&= \sum_{l=0}^{\log_b n} a^l \cdot c \cdot \frac{n^d}{b^{ld}} =\\
&= cn^d \sum_{l=0}^{\log_b n} \frac{a^l}{b^{ld}} =\\
&= cn^d \sum_{l=0}^{\log_b n} \left(\frac{a}{b^d}\right)^l =\\
&= cn^d \sum_{l=0}^{\log_b n} \alpha^l, \text{ where } \alpha = \frac{a}{b^d}. \qquad (4.5)
\end{aligned}$$

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 4.2 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

$$a^l \cdot c \cdot \left(\frac{n}{b^l}\right)^d .$$



level	size	subproblems	work
0	n	1	cn^d
1	n/b	a	$ca(\frac{n}{b})^d$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
l	n/b^l	a^l	$ca^l(\frac{n}{b^l})^d$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_b n$	1	$a^{\log_b n}$	

Figure 4.2: A tree of recursive calls of a divide-and-conquer algorithm for $a = 3$.

By taking the sum over all levels $l = 0, 1, \ldots, \log_b n$, we get (4.4). Thus, it remains to estimate the order of growth of (4.5).

The sum (4.5) 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 (4.2)). 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:

- $T(n) \le 5T(n/4) + O(n)$
- $T(n) \le 5T(n/4) + O(n^2)$
- $T(n) \le 7T(n/2) + O(n^2)$

To conclude, let us show a few recurrences that are not covered (at least, directly) by the Master Theorem.

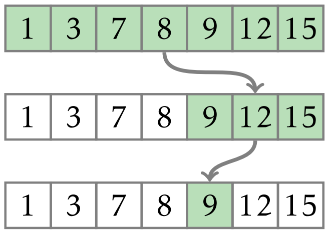
- $T(n) \le T(n/3) + T(2n/3) + O(n)$. In this case, there are two recursive calls, but for problems of *different* size. In particular, the tree of recursive calls is going to be imbalanced. Still, by essentially the same analysis one can show that $T(n) \le O(n \log n)$: though the tree is imbalanced, its depth is logaritmic, and the work at every level is still $O(n)$.
- $T(n) \le 2T(n-1) + O(1)$. In this case, the size of subproblems is not of the form n/b. It is possible to show that $T(n) \le O(2^n)$ 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.

4.2 Programming Challenges

4.2.1 Binary Search

Sorted Array Search Problem
Search a key in a sorted array of keys.

Input: A sorted array $K = [k_0, \ldots, k_{n-1}]$ of distinct integers (i.e., $k_0 < k_1 < \cdots < k_{n-1}$) 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 *minIndex* equal to 0 and *maxIndex* equal to $n-1$. It sets *midIndex* to $(minIndex+maxIndex)/2$ and then checks to see whether q is greater than or less than $K[midIndex]$. If q is larger than this value, then BinarySearch iterates on the subarray of K from *minIndex* to $midIndex - 1$; otherwise, it iterates on the subarray of K from $midIndex + 1$ to *maxIndex*. Iteration eventually identifies whether q occurs in K.

```
BinarySearch($K[0..n-1], q$)
$minIndex \leftarrow 0$
$maxIndex \leftarrow n-1$
while $maxIndex \geq minIndex$:
  $midIndex \leftarrow \lfloor (minIndex + maxIndex)/2 \rfloor$
  if $K[midIndex] = q$:
    return $midIndex$
  else if $K[midIndex] < q$:
    $minIndex \leftarrow midIndex + 1$
  else:
    $maxIndex \leftarrow midIndex - 1$
return $-1$
```

For example, if $q = 9$ and $K = [1, 3, 7, 8, 9, 12, 15]$, BinarySearch would first set *minIndex* = 0, *maxIndex* = 6, and *midIndex* = 3. Since q is greater than $K[\mathit{midIndex}] = 8$, we examine the subarray whose elements are greater than $K[\mathit{midIndex}]$ by setting *minIndex* = 4, so that *midIndex* is recomputed as $(4+6)/2 = 5$. This time, q is smaller than $K[\mathit{midIndex}] = 12$, 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 $Q = [q_0, \ldots, q_{m-1}]$ 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 $k_0 < k_1 < \ldots < k_{n-1}$ of n distinct positive integers in increasing order. The next two lines contain an integer m and m positive integers $q_0, q_1, \ldots, q_{m-1}$.

Output format. For all i from 0 to $m-1$, output an index $0 \le j \le n-1$ such that $k_j = q_i$, or -1, if there is no such index.

Constraints. $1 \le n \le 3 \cdot 10^4$; $1 \le m \le 10^5$; $1 \le k_i \le 10^9$ for all $0 \le i < n$; $1 \le q_j \le 10^9$ for all $0 \le j < m$.

Sample.

Input:

```
5
1 5 8 12 13
5
8 1 23 1 11
```

Output:

```
2 0 -1 0 -1
```

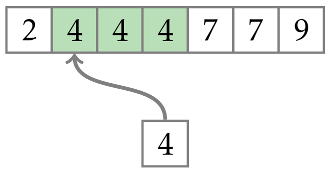
Queries 8, 1, and 1 occur at positions 3, 0, and 0, respectively, while queries 23 and 11 do not occur in the sequence of keys.

4.2.2 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 $k_0 \le k_1 \le \cdots \le k_{n-1}$ of n positive integers in non-decreasing order. The next two lines contain an integer m and m positive integers $q_0, q_1, \ldots, q_{m-1}$.

Output format. For all i from 0 to $m-1$, output the index $0 \le j \le n-1$ of the first occurrence of q_i (i.e., $k_j = q_i$) or -1, if there is no such index.

Constraints. $1 \le n \le 3 \cdot 10^4$; $1 \le m \le 10^5$; $1 \le k_i \le 10^9$ for all $0 \le i < n$; $1 \le q_j \le 10^9$ for all $0 \le j < m$.

Sample.

Input:

```
7
2 4 4 4 7 7 9
4
9 4 5 2
```

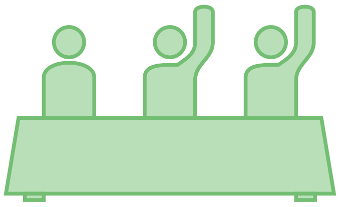
Output:

```
6 1 -1 0
```

4.2.3 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. $a_0, \ldots, a_{n-1}$.

Output format. Output 1 if the sequence contains an element that appears more than $n/2$ times, and 0 otherwise.

Constraints. $1 \le n \le 10^5$; $0 \le a_i \le 10^9$ for all $0 \le i < n$.

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:

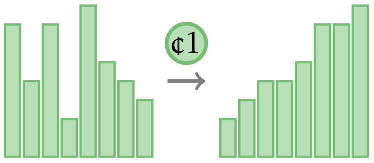
```
0
```

This sequence does not have a majority element (note that the element 1 is not a majority element).

4.2.4 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.



Recall 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:

$$a \cdot n + a \cdot (n-1) + a \cdot (n-2) + \cdots = a \cdot \frac{n \cdot (n+1)}{2},$$

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 $a_0, a_1, \ldots, a_{n-1}$.

Output format. Output this sequence sorted in non-decreasing order.

Constraints. $1 \le n \le 10^5$; $1 \le a_i \le 10^9$ for all $0 \le i < n$.

Sample.

Input:

```
5
2 3 9 2 2
```

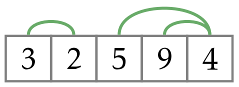
Output:

```
2 2 2 3 9
```

4.2.5 Number of Inversions

Number of Inversions Problem
Compute the number of inversions in a sequence of integers.

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

Output format. The number of inversions in the sequence.

Constraints. $1 \le n \le 30\,000$, $1 \le a_i \le 10^9$ for all $1 \le i \le n$.

Sample.

Input:

```
5
2 3 9 2 9
```

Output:

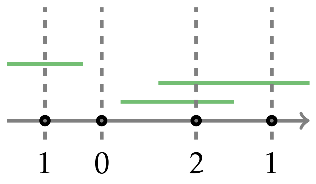
```
2
```

The two inversions here are $(2,4)$ ($a_2 = 3 > 2 = a_4$) and $(3,4)$ ($a_3 = 9 > 2 = a_4$).

4.2.6 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 $p_1, \ldots, p_m$.

Output format. m non-negative integers $k_1, \ldots, k_p$ where k_i is the number of segments that contain p_i.

Constraints. $1 \le n, m \le 50\,000$; $-10^8 \le l_i \le r_i \le 10^8$ for all $1 \le i \le n$; $-10^8 \le p_j \le 10^8$ for all $1 \le j \le m$.

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
```

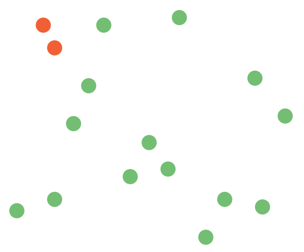
Output:

```
2 0
```

4.2.7 Closest Points

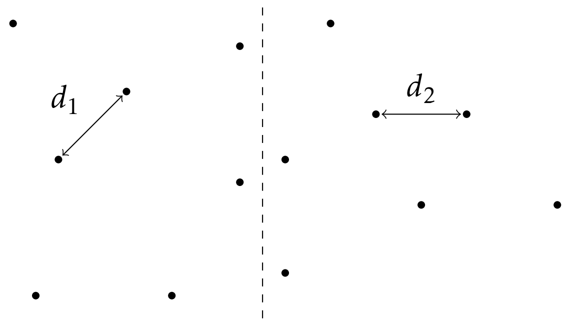
Closest Points Problem
Find the closest pair of points in a set of points on a plane.

> **Input:** A list of n points on a plane.
> **Output:** The minimum distance between a pair of these points.

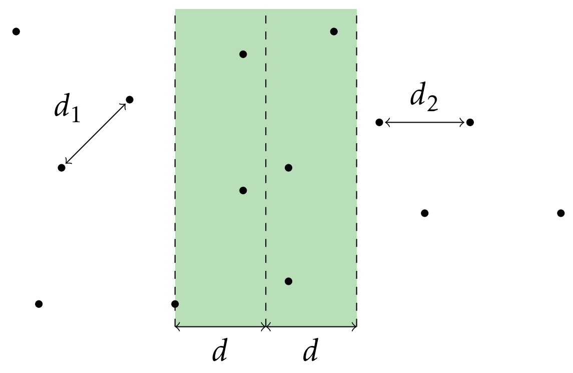


This computational geometry problem has many applications in computer graphics and vision. A naive algorithm with quadratic running time iterates through all pairs of points to find the closest pair. Your goal is to design an $O(n \log n)$ time divide and conquer algorithm.

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 $d = \min\{d_1, d_2\}$.



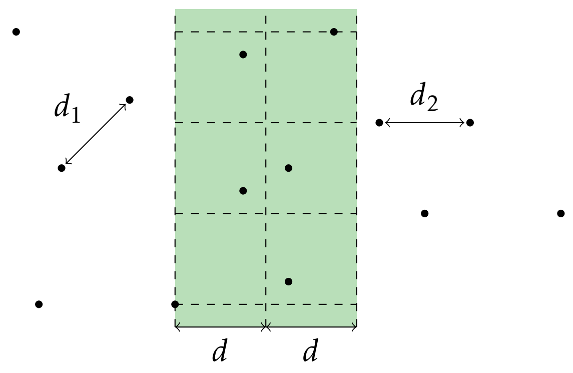
It remains to check whether there exist points $p_1 \in S_1$ and $p_2 \in S_2$ such that the distance between them is smaller than d. We cannot afford to check all possible such pairs since there are $\frac{n}{2} \cdot \frac{n}{2} = \Theta(n^2)$ of them. To check this faster, we first discard all points 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 $P = [p_1, \ldots, p_k]$. 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. $2 \le n \le 10^5$; $-10^9 \le x_i, y_i \le 10^9$ are integers.

Sample 1.

Input:

```
2
0 0
3 4
```

Output:

```
5.0
```

There are only two points at distance 5.

Sample 2.

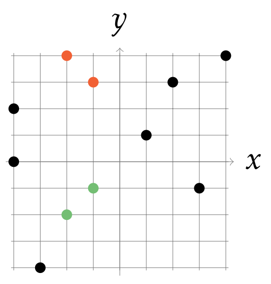
Input:

```
11
4 4
-2 -2
-3 -4
-1 3
2 3
-4 0
1 1
-1 -1
3 -1
-4 2
-2 4
```

Output:

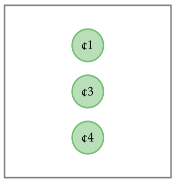
```
1.414213
```

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 $(-1,3)$.



Chapter 5: 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.

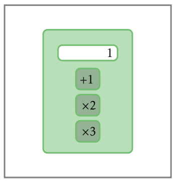


Money Change Again

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*.

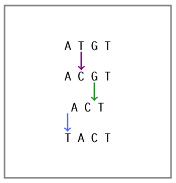


Primitive Calculator

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”, “×2”, and “×3” needed to get n from 1.

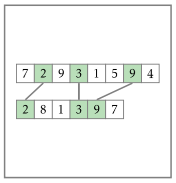


Edit Distance

Compute the edit distance between two strings.

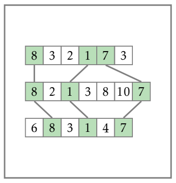
Input. Two strings.

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



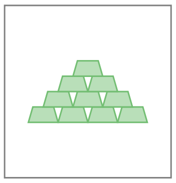
Longest Common Subsequence of Two Sequences

Compute the maximum length of a common subsequence of two sequences.
Input. Two sequences.
Output. The maximum length of a common subsequence.



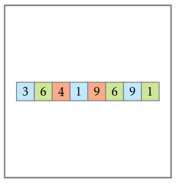
Longest Common Subsequence of Three Sequences

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 $w_1,\ldots,w_n$ 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 $v_1, v_2, \ldots, v_n$.
Output. Check whether it is possible to partition them into three subsets with equal sums, i.e., check whether there exist three disjoint sets $S_1, S_2, S_3 \subseteq \{1,2,\ldots,n\}$ such that $S_1 \cup S_2 \cup S_3 = \{1,2,\ldots,n\}$ and

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

$$\begin{tikzcd}
((8-5)\times 3)=9 \\
8-5\times 3 \arrow[u] \arrow[d] \\
(8-(5\times 3))=-7
\end{tikzcd}$$

Maximum Value of an Arithmetic Expression

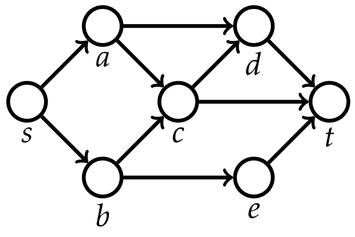
Parenthesize an arithmetic expression to maximize its value.
Input. An arithmetic expression consisting of digits as well as plus, minus, and multiplication signs.
Output. Add parentheses to the expression in order to maximize its value.

5.1 The Main Idea

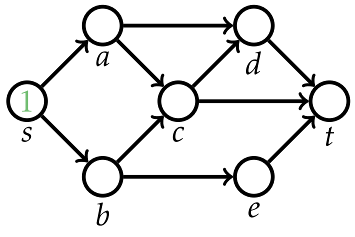
5.1.1 Number of Paths

To "invent" the key idea of the dynamic programming technique, try to solve the following puzzle.

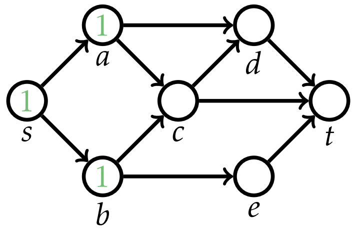
Interactive Puzzle "Number of Paths". There are many ways of getting from s to t in the network below: for example, $s \to b \to e \to t$ and $s \to a \to c \to d \to t$. What is the total number of paths? Try it online (level 1)!



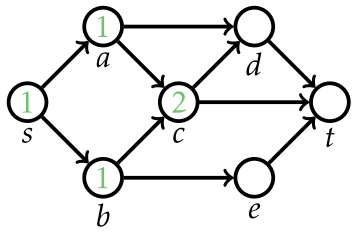
Since we start from s, there is a unique way to get to s. Let's write this down:



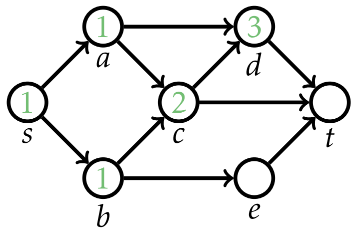
For a and b, there is also just a single path.



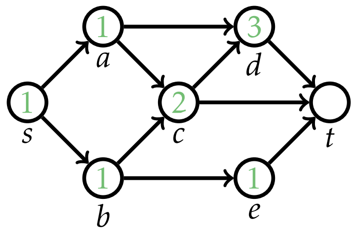
Since there is only one path to a and only one path to b, the number of paths to c is $1 + 1 = 2$ ($s \to a \to c$ and $s \to b \to c$).



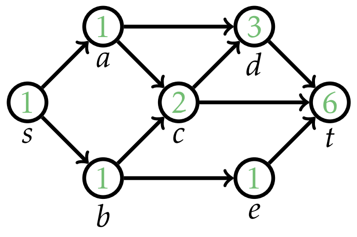
Similarly, to get to d one needs to get to either a or c. There is one path to get to a and two paths to get to c. Hence, the number of paths to get to d is $1+2=3$ ($s \to a \to d$, $s \to a \to c \to d$, and $s \to b \to c \to d$).



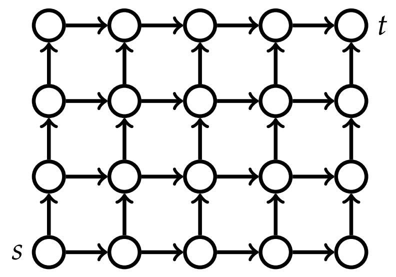
The number of paths ending in e is equal to 1 as e can be reached from b only.

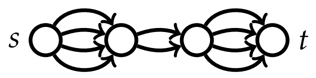


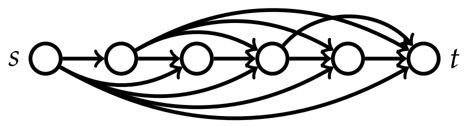
Since there are two paths to c, three paths to d, and one path to e, there are $2+3+1=6$ paths to t.



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 5×4 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?

5.1.2 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 $paths(v)$ be the number of paths from s to a node v. Clearly, $paths(s) = 1$. This is called a *base case*. For all other nodes, the corresponding value can be found using a *recurrence relation*:

$$paths(v) = \sum_{\text{each predecessor } w \text{ of } v} paths(w)\,,$$

where a predecessor of v is a node that has an edge connecting it with v.

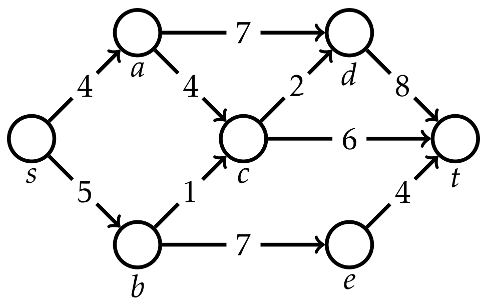
Many dynamic programming algorithms follow the same pattern:

- 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.

5.1.3 Shortest Path in Directed Acyclic Graph

Now, consider a *weighted graph* where each edge e has length denoted *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 $s \to b \to e \to t$ 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,

$$\mathit{length}(t) = \min\{\mathit{length}(c) + 6, \mathit{length}(d) + 8, \mathit{length}(e) + 4\},$$

where *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:

$$\begin{aligned}\mathit{length}(c) &= \min\{\mathit{length}(a) + 4, \mathit{length}(b) + 1\},\\ \mathit{length}(d) &= \min\{\mathit{length}(a) + 7, \mathit{length}(c) + 2\},\\ \mathit{length}(e) &= \mathit{length}(b) + 7.\end{aligned}$$

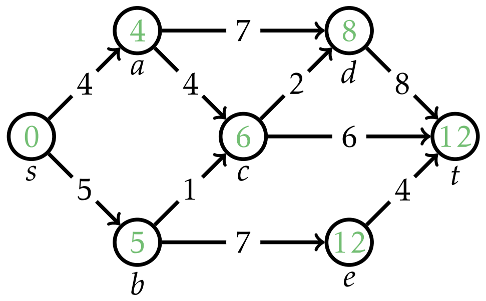
The recurrence relations for a and b are the following:

$$\begin{aligned}\mathit{length}(a) &= \mathit{length}(s) + 4,\\ \mathit{length}(b) &= \mathit{length}(s) + 5.\end{aligned}$$

Finally, the base case is *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:

$$length(v) = \min_{\text{each predecessor } w \text{ of } v} \{length(w) + length(w,v)\}.$$

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 $length(t)$:

$$length(t) = \min\{length(c) + 6, length(d)+8, length(e)+4\} = \min\{12, 16, 16\} = 12.$$

From this, we conclude that the last edge of an optimum path is $c \to t$. Similarly,

$$length(c) = \min\{length(a) + 4, length(b) + 1\} = \min\{8, 6\} = 6,$$

hence, we arrive to c from b. Thus, the path from s to t of length 12 is

$$s \to b \to c \to t.$$

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.

5.2 Programming Challenges

5.2.1 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. $1 \le money \le 10^3$.

Sample.

Input:

```
34
```

Output:

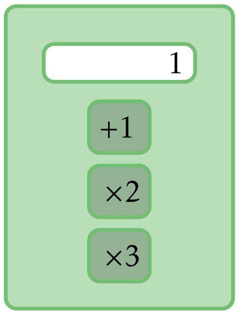
```
9
```

$34 = 3+3+4+4+4+4+4+4+4$.

5.2.2 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

$$\textsc{GreedyCalculator}(n)$$

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 $a_0, a_1, \ldots, a_k$ such that $a_0 = 1$, $a_k = n$ and for all $1 \le i \le k$, 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. $1 \le n \le 10^6$.

Sample 1.

Input:

```
1
```

Output:

```
0
1
```

Sample 2.

Input:

```
96234
```

Output:

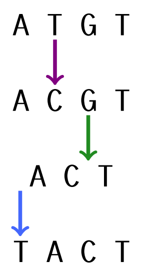
```
14
1 3 9 10 11 22 66 198 594 1782 5346 16038 16039 32078 96234
```

Another valid output in this case is “1 3 9 10 11 33 99 297 891 2673 8019 16038 16039 48117 96234”.

5.2.3 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
```

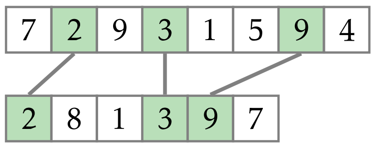
Output:

```
0
```

5.2.4 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 $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$, their common subsequence of length p is a et of p indices

$$1 \le i_1 < i_2 < \cdots < i_p \le n,$$
$$1 \le j_1 < j_2 < \cdots < j_p \le m.$$

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: $a_1, a_2, \ldots, a_n$. Third line: m. Fourth line: $b_1, b_2, \ldots, b_m$.

Output format. p.

Constraints. $1 \le n, m \le 100$; $-10^9 \le a_i, b_i \le 10^9$ 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:

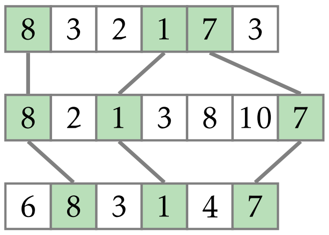
```
2
```

One common subsequence is $(2,7)$. Another one is $(2,8)$.

5.2.5 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 $A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_m)$, and $C = (c_1, c_2, \ldots, c_l)$, find the length of their longest common subsequence, i.e., the largest non-negative integer p such that there exist indices

$$\begin{aligned}
&1 \le i_1 < i_2 < \cdots < i_p \le n,\\
&1 \le j_1 < j_2 < \cdots < j_p \le m,\\
&1 \le k_1 < k_2 < \cdots < k_p \le l
\end{aligned}$$

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: $a_1, a_2, \ldots, a_n$. Third line: m. Fourth line: $b_1, b_2, \ldots, b_m$. Fifth line: l. Sixth line: $c_1, c_2, \ldots, c_l$.

Output format. p.

Constraints. $1 \le n, m, l \le 100$; $-10^9 \le a_i, b_i, c_i \le 10^9$.

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:

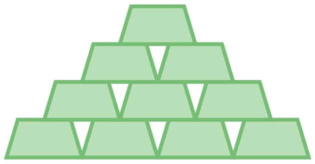
```
3
```

One common subsequence of length 3 in this case is $(8,3,7)$. Another one is $(8,1,7)$.

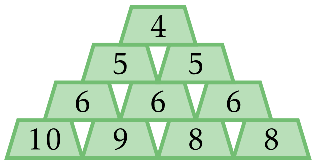
5.2.6 Maximum Amount of Gold

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 $w_1, \ldots, w_n$ 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 $w_1, \ldots, w_n$ defining the weights of the gold bars.

Output format. The maximum weight of gold bars that fits into a backpack of capacity W.

Constraints. $1 \le W \le 10^4$; $1 \le n \le 300$; $0 \le w_1, \ldots, w_n \le 10^5$.

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.

Iterative vs Recursive Dynamic Programming Algorithms

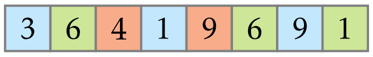
In many cases, an iterative approach is preferable since it has no recursion overhead and it may allow saving space by exploiting a regular structure of the table. Still, a recursive approach has its own advantages and may be even preferable for some problems. First, it may be faster if not all the subproblems need to be solved. For example, whereas it is clear that the dynamic programming algorithm for finding the edit distance of two strings needs to solve all subproblems (that is, to find the edit distance of all its prefixes), it is not the case for the Maximum Amount of Gold Problem. Indeed, if the weight of all bars is divisible by, say, ten, then we are just not interested in the values of $pack(w,i)$ when w is not divisible by ten. Still, our iterative algorithm for the problem computes $pack(w,i)$ for all pairs (w,i)! At the same time, the recursive algorithm computes the values of $pack(w,i)$ for only those (w,i) that are needed for computing the final value. For the special case when all weights are divisible by ten, the iterative algorithm is about ten times slower than the recursive one. Second, if recursion overhead is not that important, a recursive algorithm may be easier to implement: one just uses a hash table and converts (almost mechanically) a recurrence relation into a recursive algorithm. When doing this, one does not have to think about data structures for storing the solutions for subproblems and an order of solving subproblems.

5.2.7 Splitting the Pirate Loot

3-**Partition Problem**
Partition a set of integers into three subsets with equal sums.

Input: A sequence of integers $v_1, v_2, \ldots, v_n$.
Output: Check whether it is possible to partition them into three subsets with equal sums, i.e., check whether there exist three disjoint sets $S_1, S_2, S_3 \subseteq \{1, 2, \ldots, n\}$ such that $S_1 \cup S_2 \cup S_3 = \{1, 2, \ldots, n\}$ 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 $v_1, v_2, \ldots, v_n$ separated by spaces.

Output format. Output 1, if it possible to partition $v_1, v_2, \ldots, v_n$ into three subsets with equal sums, and 0 otherwise.

Constraints. $1 \le n \le 20$, $1 \le v_i \le 30$ 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
```

$1+3+7+25 = 2+4+5+7+8+10 = 5+12+19.$

5.2.8 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.

$$\begin{tikzcd} ((8-5)\times 3)=9 \\ 8-5\times 3 \arrow[u] \arrow[d] \\ (8-(5\times 3))=-7 \end{tikzcd}$$

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 $s_0, s_1, \ldots, s_{2n}$. 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. $0 \le n \le 14$ (hence the string contains at most 29 symbols).

Sample.

Input:

```
5-8+7*4-8+9
```

Output:

```
200
```

$200 = (5-((8+7)\times(4-(8+9))))$

5.3 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 5.2.6.)

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.

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:

```
nodejs
```

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 1.2.4.

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.