Mastering CodingBat (Java)

Books by Ulm Publishing

Mastering CodingBat (Java) Vol. 1: Basics
Mastering CodingBat (Java) Vol. 2: Intermediate A
Mastering CodingBat (Java) Vol. 3: Intermediate B
Mastering CodingBat (Java) Vol. 4: Advanced

Mastering CodingBat (Java)

Vol. 2: Intermediate A

Gregor Ulm

Ulm Publishing

Contents

Preface

In 2012, I published what was back then the first full set of solutions to CodingBat (codingbat.com), both in Java and Python, on my website gregorulm.com. To my great surprise, there has been substantial interest, judging from the number of visitors, comments, and emails I have received over the years. The solutions I wrote and published online I originally wrote down very quickly, making sure they look clean enough and pass all test cases.

Over time, I noticed that people started to view my solutions as a work of reference, comparing their solutions to mine. That was not my intention at all. However, as I am convinced of the didactic value of CodingBat, I would like to further expand on my online solutions in a series of books. The goal of these books is to introduce problems, describe how to tackle them, give hints and, which is of utmost interest to people learning how to program, present fully-worked solutions that discuss how you get from an idea to a working piece of code. That happens in a number of steps, which may include discussing alternative approaches. By doing so, a novice programmer will make two important realizations. The first one is that there are many valid solutions to a given problem, while the second one is that non-trivial programming tasks are performed incrementally, which may involve dismissing earlier attempts.

The expected reader is a novice programmer, probably a freshman in college. You may also be a high school student or a self-learner. By working through the books in the Mastering CodingBat series, the reader will gain a very thorough understanding of elementary programming concepts. This will provide an excellent preparation for future projects, either during your university studies or at work. I found it very gratifying to wit-

ness the positive reception my CodingBat solutions have found. I hope that these books find an equally warm reception.

Gregor Ulm
Gothenburg, Sweden

Introduction

The exercise set published on CodingBat.com is an excellent tool for leaning how to program. According to its creator, Stanford professor Nick Parlante, the motivation behind his site is to gradually introduce people to solving increasingly difficult problems programmatically. This is a particularly useful tool for novices because the gap from not knowing much to having to solve the kind of problems you are confronted with in a computer science degree can be quite intimidating.

Programming has become a highly useful skill. Even if you have no aspirations of becoming a full-time software developer, it is not at all implausible that your field, no matter what it is, can advance dramatically with increased automation. For the most part, that is what programmers do: they automate processes. Imagine how much more effective you could be at your job if you were able to automate (or outsource) everything that is tedious and repetitive! If you fully understand all the problems on CodingBat, you are well on the way of joining the automators, if you so desire.

With this book, the goal is to dissect every problem, as of January 2019, in the five CodingBat sections *Warmup-2*, *String-2*, *Array-2*, *Logic-2*, and `Map-1`. Those sections contain slightly more complex problems than the ones you have encountered in the first Mastering CodingBat book. Mastery of the concepts these problems test is a prerequisite for any professional software developer.

This book is primarily for everyone who has encountered CodingBat and would like to get a bit more hand-holding, better guidance, or some help with understanding the various problems. You could be a high school student, or maybe you are enrolled in college. You may even be an autodidact. No mat-

ter your role, you may have encountered the problem that even books and other materials that are supposedly written for beginners assume a significant amount of knowledge. Even worse, many of those books are incredibly, incredibly verbose. In contrast, this book presents you with the meat of programming. If you make it through it, you will become a better programmer.

How to use this book

View this book as your personal tutor who guides you through CodingBat. The presentation of each problem has a fixed format. We start with the *problem description*, which also contains a skeleton of the code, i.e. a function signature. It is your task to come up with the body of the function. There are many ways to skin a cat, and there are at least as many ways to solving problems programmatically. However, some approaches keep you from learning certain skills. Thus, this book lists the *tools* you can use for solving a given problem. It may be tempting to just use an inbuilt method in Java that does what you are supposed to write code for, but if you do that, you are not going to progress much. Unlike in the first book of this series, this time only methods and functions are mentioned in this section. I assume that you are by now very familiar with the standard operators of Java.

In order to guide you in the right directions, an optional *hints* section provides a few pointers. Depending on the problem, those can be substantial or almost nonexistent. Sometimes it is just a reference to a previous and similar problem. This is followed by the *solution*, which shows at least one possible solution to the problem, with explanations of the program logic. There may be more than one suggested solution if there are interesting alternative approaches.

Treat this book essentially as extensive commentary. I would suggest to go through it sequentially as the problems normally increase in difficulty. If you jump around and get stuck, you may only get needlessly frustrated. For each problem, start by reading the problem description. Then look at the code skeleton. Instead of writing code on the CodingBat website, however, I would like you to use *pencil and paper*. Yes, you

read that correctly. Before starting to type, write code by hand and try to come up with a solution that is as close to working code as possible. By not writing executable Java code straight away you won't get tempted to, for instance, guessing and using the website as some kind of feedback generator that helps you to gradually figure out the solution. Instead, with your pencil in hand, you figure out the problem before writing any code. If you can't do that, then you may need to think harder about the problem. As you get more experienced, you may want to skip writing code down on paper first. However, if you find yourself changing your code too much, then you probably need to spend more time figuring out the solution in your head first.

Once you are content with the code you have written down on paper, go to the CodingBat website. Enter your solution, execute the code. Fix syntactic errors if there are any. (Optional: if you get stuck, use my hints.) After you've solved the problem correctly, look at my solution and the discussion. Compare your solution with mine. Make sure you understand every single line in both your code and my solutions as well as all the variations.

Lastly, I would recommend you work steadily through this book. Do at least one or two problems every day, but ideally more. I think you should be able to do at least five problems a day very comfortably.

Warmup-2

stringTimes

Problem

The input of the function `stringTimes` consists of a string `str` and a non-negative integer `n`. The goal is to produce an output string that consists of the concatenation of `n` copies of `str`. See Listing 1 for the code skeleton.

```
1 public String stringTimes(String str, int n) {
2   // your code here
3 }
```

Listing 1: `stringTimes` – skeleton

Tools

You do not need to use any inbuilt functions for this problem.

Hints

Use an accumulator variable `acc` for building the value you want to return.

Solution

This is a very straightforward problem. Start with initializing a variable `acc` to the empty string. Afterwards, use a for-loop to concatenate the input string `str` `n` times with the string `acc`. Listing 2 shows the code.

```
1 public String stringTimes(String str, int n) {
2   String result = "";
3   for (int i = 0; i < n; i++) {
4     result += str;
5   }
6   return result;
7 }
```

Listing 2: `stringTimes` – solution (v1)

Of course, you may as well condense the for-loop to just one line, as shown in Listing 3.

```
1 public String stringTimes(String str, int n) {
2   String acc = "";
3   for (int i = 0; i < n; i++) acc += str;
4   return acc;
5 }
```

Listing 3: `stringTimes` – solution (v2)

frontTimes

Problem

The function `frontTimes` takes a string `str` and a non-negative integer `n` as its input. The goal is to return a string that consists of a concatenation of the *front* of `str`. In this case, the front consists of the first three characters of `str`, or the entire string, if its length is less than 3. See Listing 4 for the code skeleton.

```
1 public String frontTimes(String str, int n) {
2   // your code here
3 }
```

Listing 4: `frontTimes` – skeleton

Tools

You may want to use the string methods `length` and `substring`.

Hints

First, determine which characters constitute the front. Afterwards, build the string you want to return.

Solution

This problem is an extension of the previous one. An obvious solution is to make a case distinction based on the length of the input string `str`. If `str` is at most 3 characters long, build the eventual return value using the entire string. Otherwise, use a substring that is made of the first three characters of `str`. This approach is shown in Listing 5.

```
1  public String frontTimes(String str, int n) {
2    String acc = "";
3    if (str.length() <= 3) {
4      for (int i = 0; i < n; i++) {
5        acc += str;
6      }
7    } else {
8      String front = str.substring(0, 3);
9      for (int i = 0; i < n; i++) {
10       acc += front;
11     }
12   }
13   return acc;
14 }
```

Listing 5: `frontTimes` – solution (v1)

As you can probably see, there is a bit of redundancy as the two for-loops are similar. This could be resolved by adding a helper function, but that would not make it any less awkward. Instead, we could determine the front of the string with a conditional expression and consequently eliminate one of the for-loops. This is shown in Listing 6. The ternary operator in line 3 is used for conciseness.

```
1  public String frontTimes(String str, int n) {
2    String acc = "";
3    int pos = str.length() <= 3 ? str.length() : 3;
4    String front = str.substring(0, pos);
5    for (int i = 0; i < n; i++) {
6      acc += front;
7    }
8    return acc;
9  }
```

Listing 6: `frontTimes` – solution (v2)

String-2

catDog

Problem

The function `catDog` takes a string `str` as its input and returns `true` if the substring `"cat"` appears in it as often as the substring `"dog`. Otherwise, it returns `false`. See Listing 46 for the code skeleton.

```
1 public boolean catDog(String str) {
2   // your code here
3 }
```

Listing 46: `catDog` – skeleton

Tools

You may want to use the string methods `length`, `substring`, and `equals`.

Hints

While an obvious solution would maintain two separate counters for the occurrences of the respective substring, think about whether you can come up with a more elegant approach.

Solution

The first version of our solution uses two different accumulators for the respective occurrences of the substrings `"cat"` and `"dog`. In the for-loop, we iterate through the entire string, keeping the boundaries of the array in mind, and increment the accumulators whenever we encounter one of those substrings. Afterwards, we check if both accumulators contain the same number, and return the corresponding boolean value. See Listing 47 for the code.

```
1  public boolean catDog(String str) {
2    int cats = 0;
3    int dogs = 0;
4    for (int i = 0; i < str.length()-2; i++) {
5      String tmp = str.substring(i, i+3);
6      if (tmp.equals("cat")) cats += 1;
7      if (tmp.equals("dog")) dogs += 1;
8    }
9    return cats == dogs;
10 }
```

Listing 47: `catDog` – solution (v1)

There is some redundancy in the previous code because we do not need to count how often we encounter each of those substrings. Instead, it is sufficient to use only one variable and increment it whenever we encounter `"cat"`, and decrement it when we encounter `"dog"`. At the end, we return `true` if the counter is equal to 0, and `false` otherwise. See Listing 48 for the code.

```
1  public boolean catDog(String str) {
2    int count = 0;
3    for (int i = 0; i < str.length()-2; i++) {
4      String tmp = str.substring(i, i+3);
5      if (tmp.equals("cat")) count += 1;
6      if (tmp.equals("dog")) count -= 1;
7    }
8    return count == 0;
9  }
```

Listing 48: `catDog` – solution (v2)

Of course, for a small problem such as this one, the difference is minor. However, as your programming projects grow larger,

you will find it increasingly helpful to limit superfluous variables (or state, for functional programmers).

xyBalance

Problem

The function `xyBalance` takes a string `str` as its input. If there is at least one `'x'` character in `str`, return `true` if it is the case that once a `'y'` character appears in the string there are no more `'x'` characters. Return `false` otherwise. See Listing 64 for the code skeleton.

```
1 public boolean xyBalance(String str) {
2   // your code here
3 }
```

Listing 64: `xyBalance` – skeleton

Tools

There is a very easy way to solve this problem, and one way that teaches you a bit more computational thinking. For the easy option, use the string method `lastIndexOf`. For more of a challenge, do not use that string method but instead only use the string methods `length` and `charAt`.

Hints

If you want to get the most out of this problem then do not use `lastIndexOf`.

Solution

If you want to use the string method `lastIndexOf`, you only have to compare the two positions for the characters `'x'` and `'y'`. Note that this method returns the value `-1` if the character searched for has not been found. Thus, we can turn the solution into a one-liner. See Listing 65 for the code.

```
1 public boolean xyBalance(String str) {
2   return str.lastIndexOf("x") <= str.lastIndexOf("y");
3 }
```

Listing 65: `xyBalance` – solution (v1)

Of course, resorting to an inbuilt function like that is only an effective way for you to cheat yourself out of honing your programming skills. Thus, we will continue with a different solution. For this one, we first determine the position of the last `'y'` with a for-loop. As a second step, we look through `str` again and determine if the last occurrence of `'x'` is after the last occurrence of `'y'`. See Listing 66 for the code.

```
1  public boolean xyBalance(String str) {
2    int lastY = -1;
3    for (int i = 0; i < str.length(); i++) {
4      if (str.charAt(i) == 'y') lastY = i;
5    }
6    for (int i = 0; i < str.length(); i++) {
7      if (str.charAt(i) == 'x') {
8        if (i > lastY) return false;
9      }
10   }
11   return true;
12 }
```

Listing 66: `xyBalance` – solution (v2)

There is a more elegant way of solving this problem, namely by going through `str` from the back. If we encounter a `'y'` after having encountered an `'x'`, we return `false`. Otherwise, we return `true`. See Listing 67 for the code.

```
public boolean xyBalance(String str) {
  boolean seen_x = false;
  for (int i = str.length() - 1; i >= 0; i--) {
    if (str.charAt(i) == 'x') seen_x = true;
    if (str.charAt(i) == 'y') {
      if (seen_x) return false;
    }
  }
  return true;
}
```

Listing 67: `xyBalance` – solution (v3)

Array-2

countEvens

Problem

The function `countEvens` takes an array of integers `nums` as its input. It returns an integer indicating how many even integers there are in that array. See Listing 100 for the code skeleton.

```
1 public int countEvens(int[] nums) {
2   // your code here
3 }
```

Listing 100: `countEvens` – skeleton

Tools

You will need to use the array attribute `length`.

Hints

A number `x` is odd if it is true that `x` modulus 2 is equal to 1. Expressed in Java, this becomes `x % 2 == 1`.

Solution

Initialize an accumulator `count` to 0. Afterwards, iterate through `nums` and check, for every element, if it is even by taking its modulus 2. Whenever you encounter an odd number, increment `count`. See Listing 101 for the code.

```
public int countEvens(int[] nums) {
  int count = 0;
  for (int i = 0; i < nums.length; i++)
    if (nums[i] % 2 == 0) count++;
  return count;
}
```

Listing 101: `countEvens` – solution

bigDiff

Problem

The function `bigDiff` takes an array of integers `nums` as its input, which contains at least one element. It returns the difference between its largest and smallest value. See Listing 102 for the code skeleton.

```
public int bigDiff(int[] nums) {
  // your code here
}
```

Listing 102: `bigDiff` – skeleton

Tools

Use standard array operators for this problem. The CodingBat website also recommends using the inbuilt methods `Math.max` and `Math.min`, but try to also solve this problem without them.

Solution

The input array `nums` contains at least one element. Thus, we can declare two variables `max` and `min`, and initialize them to `nums[0]`. Afterwards, we iterate through `nums` and compare each element with both `max` and `min`. If the current element is larger than the current value of `max`, we update that variable. Similarly, if the current element is larger than the current value of `min`, we update `min`. At the end, we return the difference between `max` and `min`. See Listing 103 for the code.

```
1 public int bigDiff(int[] nums) {
2   int max = nums[0];
3   int min = nums[0];
4   for (int i = 0; i < nums.length; i++) {
5     if (nums[i] >  max) max = nums[i];
6     if (nums[i] <= min) min = nums[i];
7   }
8   return max - min;
9 }
```

Listing 103: `bigDiff` – solution (v1)

Alternatively, you could use `Math.max` and `Max.min` to iteratively update the variables `max` and `min`. To do so, determine the maximum as well as the minimum of the current element and `max` and `min`, respectively. See Listing 104 for the code.

```
1 public int bigDiff(int[] nums) {
2   int max = nums[0];
3   int min = nums[0];
4   for (int i = 0; i < nums.length; i++) {
5     max = Math.max(nums[i], max);
6     min = Math.min(nums[i], min);
7   }
8   return max - min;
9 }
```

Listing 104: `bigDiff` – solution (v2)

Logic-2

makeBricks

Problem

The function `makeBricks` takes three integers `small`, `big`, and `goal` as its input. The first value stands for the amount of bricks that are 1 inch long, the second for the amount of bricks that are 5 inches long, and the third is the length we want to achieve by placing those bricks one after the other. Return `true` if the goal can be achieved and `false` otherwise. See Listing 177 for the code skeleton.

```
1 public boolean makeBricks(int small, int big, int goal) {
2   // your code here
3 }
```

Listing 177: `makeBricks` – skeleton

Tools

Using `Math.min` may be helpful. However, this problem can be solved just with standard Java operators.

Solution

Let us start with `goal` and see if we can reduce it to 0 or less. First, we subtract from that variable as many of the big bricks times 5 as possible. This is expressed as the minimum of two values: the number of big bricks as well as the result of dividing `goal` by 5. It may be that the latter is less than the number of big bricks available, in which case we have big bricks left over, which we cannot use. The remaining value, which is now stored in `goal`, has to be reached with the available small bricks. Thus, it is sufficient to check if `goal` is less than or equal to `small`. See Listing 178 for the code.

```
1 public boolean makeBricks(int small, int big, int goal) {
2   goal -= 5 * Math.min(big, (goal / 5));
3   return goal <= small;
4 }
```

Listing 178: `makeBricks` – solution (v1)

An alternative solution works without taking the minimum. It uses two checks. First, we check if the remainder we have to cover after using up the big bricks is less than or equal to `small`. Then, we take `goal` modulus 5 and check if the result is less than or equal to `small`. See Listing 179 for the code.

```
1 public boolean makeBricks(int small, int big, int goal) {
2   return goal - big * 5 <= small
3          && goal % 5 <= small;
4 }
```

Listing 179: `makeBricks` – solution (v2)

Map-1

mapBully

Problem

The function `mapBully` takes a map `map` as its input whose keys and values are strings. It modifies `map` as follows: if it contains a key `"a"`, set the key `"b"` to that value and `"a"` to the empty string. See Listing 202 for the code skeleton.

```
public Map<String, String> mapBully(Map<String, String> map) {
  // your code here
}
```

Listing 202: `mapBully` – skeleton

Tools

You will need to use the map methods `containsKey`, `get`, and `put`.

Hints

You do not need to check if the key `"b"` is contained in `map`. If it is not, we create that key/value pair. Otherwise, we replace whatever value that key is stored with.

Solution

We check if `map` contains the key `"a"`. If so, we retrieve its value and store it in a temporary variable `tmp`. Then we store the empty string with the key `"a"` and `tmp` with the key `"b"`. See Listing 203 for the code.

```
1 public Map<String, String> mapBully(Map<String, String> map) {
2   if (map.containsKey("a")) {
3     String tmp = map.get("a");
4     map.put("a", "");
5     map.put("b", tmp);
6   }
7   return map;
8 }
```

Listing 203: `mapBully` – solution