How a Googler solves coding problems

blog.usejournal.com/how-a-googler-solves-coding-problems-ec5d59e73ec5

In this article, I'll walk you through my strategy for solving programming problems from start to finish, which I use both in my daily work at Google and with programmers of all levels (bootcampers, university students, and interns alike) as they learn and grow. Applying this structured process minimizes the frustrating debugging process and leads to cleaner, more correct code in less time.

Step By Step

I'll be using an example practice problem to illustrate.

Problem: "Given two strings, `sourceString` and `searchString`, return the first index at which `searchString` occurs within `sourceString`. If `searchString` does not occur within `sourceString`, return `-1`.

1) Draw it.

Trying to start by immediately writing code is frankly a ridiculous and lazy idea. Before you write an essay, you start by figuring out a hypothesis and evidence and ensuring that the argument makes sense. If you don't, you'll waste time starting over later when you realize what you've written doesn't fit together cohesively. Code is the same way, except worse. Like, the rubbing-shampoo-in-your-eyes kind of worse.

Often the solution to a problem is not trivial, even if it appears simple at first glance. Working it out on paper allows you to figure out a solution and verify that the solution works in a few different situations, all before even writing a single line of code.

So don't write code. Don't even think about code. You'll have plenty of time to add the semicolons and parentheses later. Just try to figure out how you, as a human computer, solve the problem.

Draw pictures. Use arrows. Put numbers into little boxes. Whatever helps you visualize the problem, do **that**. The goal is to problem-solve, and you have all the freedom of paper and pencil, with none of the constraints of

a keyboard.

Start by inventing some simple inputs. If the function “takes a string”, `"abc"` makes a great first example. Figure out what the correct result should be. Then, try to think about *how* you figured out the problem, and the steps that were involved.

Let’s imagine the strings have these values:

```
sourceString: "abcdyesefgh"
searchString: "yes"
```

My thoughts, verbatim:
Okay, so I can see that `searchString` is inside of `sourceString`. But how did I do that? Well, I started at the beginning of `sourceString` and read through it until I reached the end, looking at every 3-character piece to see if it matched the word `"yes"`. For example, `"abc"`, `"bcd"`, `"cde"`, and so on. When I got to index 4, I found `"yes"`, and so I decided that there is a match, and it starts at index 4.

When we write down our algorithm, we need to make sure we express everything and handle all the possible scenarios. Returning the right answer when we DO find a match is nice, but we also need to return the right answer when we DON’T find a match.

Let’s try again with another pair of strings:

```
sourceString: "abcdyefg"
searchString: "yes"
```

Here, we started at the beginning of `sourceString` and read through it until we reached the end, looking at every 3-character piece to see if it matched the word `"yes"`. When we got to index 4, we found `"yef"`, which was almost a match, but it wasn’t completely a match, since the third character was different. So we kept going until we reached the end of the string, and then decided there wasn’t a match, so we returned `-1`.

We’ve identified the series of steps (in programming, we call this an **algorithm**) that we take to solve the problem, and we’ve tried it in a couple of different scenarios, getting the correct result each time. At this

point, we can have some confidence that our algorithm works, and so now it's time to formalize the algorithm, bringing us to the next step:

2) Write it in English.

Here, we think about the algorithm I identified in step 1), and try to write it out in English. This makes the steps concrete, so that we can refer back to it later when writing the code.

1. Start at the beginning of the string.
2. Look at each set of 3 characters (or however many characters there are in `searchString`).
3. If any of them are equal to `searchString`, return the current index.
4. If we get to the end of the string without anything matching, return `-1`.

Seems good!

3) Write pseudocode.

Pseudocode isn't really code, but it mimics the structure of code. Here's how I would write a pseudocode version of my algorithm above:

```
for each index in sourceString,
    there are N characters in searchString
    let N chars from index onward be called POSSIBLE_MATCH
    if POSSIBLE_MATCH is equal to searchString, return index
at the end, if we haven't found a match yet, return -1.
```

I could go a bit closer to code by writing it this way:

```
for each index in sourceString,
    N = searchString.length
    POSSIBLE_MATCH = sourceString[index to index+N]
    if POSSIBLE_MATCH === searchString:
        return index
return -1
```

How much your pseudocode resembles code is up to you, and over time you'll discover the way that works best for you!

4) Translate what you can to code.

Note: For easier problems, this can be combined with the step above.

This is the first time in the process that we have to worry about syntax, function parameters and language rules. Maybe you can't write everything out, and that's okay. Write in the pieces that you know!

```
function findFirstMatch(searchString, sourceString) {
   let length = searchString.length;
   for (let index = 0; index < sourceString.length; index++) {
      let possibleMatch = <the LENGTH chars starting at index i>
      if (possibleMatch === searchString) {
         return index;
      }
   }
   return -1;
}
```

Notice that I left part of this code blank. This is intentional! I wasn't sure of the syntax for slicing strings in JavaScript, so I'll go look it up in the next step.

5) Don't guess.

A common mistake I see in new coders is the practice of finding something on the internet, saying "maybe this will work", and plugging it into your program without testing it. The more pieces of your program you don't understand, the less likely you are to end up with the right solution.

The number of ways that your program can be incorrect DOUBLES with every new thing you are not sure about. Not sure about 1 thing? Great—if your code doesn't work, only 1 thing can be the culprit.

But 2 things? 3 possibilities (thing A is broken, or thing B is broken, or both are broken!). 3 things? 7 possibilities. It quickly spirals out of control.

Side note: The formula for ways in which your program can be wrong follows the Mersenne sequence. `a(n) = (2^n)—1`

Test your new code first. Finding something on the internet is great, but before you plug it into your program, test it in a small, separate space to make sure it works in the way that you think it does.

In the previous step, I wasn't sure of the way to select a certain part of a string in JavaScript. So off I go to Google:

The first result is from w3schools. A bit dated, but usually dependable. https://www.w3schools.com/jsref/jsref_substr.asp

Based on this, I assume that I should use `substr(index, searchString.length)` to extract the portion of `sourceString` each time. But it's an **assumption**, nothing more. So first, I create a small example to test the behavior.

```
>> let testStr = "abcdefghi"
>> let subStr = testStr.substr(3, 4);  // simple, easy usage
>> console.log(subStr);
"defg"
>> subStr = testStr.substr(8, 5);   // ask for more chars than exist
"i"
```

Now I am **sure** of how this function behaves. So when I plug this into my program, I know that if my program doesn't work, it's not the new piece I added that's misbehaving.

And with that, I can plug in the final piece of my program.

```
function findFirstMatch(searchString, sourceString) {
    let length = searchString.length;
    for (let index = 0; index < sourceString.length; index++) {
        let possibleMatch = (
            sourceString.substr(index, length));
        if (possibleMatch === searchString) {
            return index;
        }
    }
    return -1;
}
```

Conclusion

If you've read to the end, all I can say now is: try it. Go back to that programming problem that you set aside last week in frustration. I guarantee you'll see immediate improvement.

Good luck, and happy coding!