10 Steps to Solving a Programming Problem

Tips for new developers staring at a blank screen, unsure of where to start

[](https://codeburst.io/@valindachan?source=post_header_lockup)

[Valinda Chan](https://codeburst.io/@valindachan)

Aug 1, 2017

Some of the feedback I hear from new developers working on a programming problem revolves around uncertainty of where to start. You understand the problem, the logic, basics of the syntax, etc. If you see someone else’s code or have someone to guide you, you can follow along. But maybe you feel uncertain about doing it yourself and have trouble turning your thoughts into code at first even though you understand the syntax or logic. **Here’s my process and some tips to tackling a sample problem that hopefully some of you may find helpful in your journey.**

**1. Read the problem at least three times (or however many makes you feel comfortable)**

**You can’t solve a problem you don’t understand.**There is a difference between the problem and the problem you think you are solving. It’s easy to start reading the first few lines in a problem and assume the rest of it because it’s similar to something you’ve seen in the past. If you are making even a popular game like Hangman, be sure to read through any rules even if you’ve played it before. I once was asked to make a game like Hangman that I realized was “Evil Hangman” only after I read through the instructions (it was a trick!).

Sometimes I’ll even try explaining the problem to a friend and see if her understanding of my explanation matches the problem I am tasked with. You don’t want to find out halfway through that you misunderstood the problem. Taking extra time in the beginning is worth it. The better you understand the problem, the easier it will be to solve it.

Let’s pretend we are creating a simple function selectEvenNumbers that will take in an array of numbers and return an array evenNumbers of only even numbers. If there are no even numbers, return the empty array evenNumbers.

function selectEvenNumbers() {  
 // your code here  
}

Here are some questions that run through my mind:

* **How can a computer tell what is an even number?** Divide that number by 2 and see if its remainder is 0.
* **What am I passing into this function?** An array
* **What will that array contain?**One or more numbers
* **What are the data types of the elements in the array?** Numbers
* **What is the goal of this function?** **What am I returning at the end of this function?** The goal is to take all the even numbers and return them in an array. If there are no even numbers, return an empty array.

**2. Work through the problem manually with at least three sets of sample data**

Take out a piece of paper and work through the problem manually. Think of at least three sets of sample data you can use. Consider [corner](https://en.wikipedia.org/wiki/Corner_case) and [edge cases](https://en.wikipedia.org/wiki/Edge_case) as well.

[*Corner case*](https://en.wikipedia.org/wiki/Corner_case)*: a problem or situation that occurs outside of normal operating parameters, specifically when multiple environmental variables or conditions are simultaneously at extreme levels, even though each parameter is within the specified range for that parameter.*

[*Edge case*](https://en.wikipedia.org/wiki/Edge_case)*: problem or situation that occurs only at an extreme (maximum or minimum) operating parameter*

For example, below are some sets of sample data to use:

[1]  
[1, 2]  
[1, 2, 3, 4, 5, 6]  
[-200.25]  
[-800.1, 2000, 3.1, -1000.25, 42, 600]

**When you are first starting out, it is easy to gloss over the steps.**Because your brain may already be familiar with even numbers, you may just look at a sample set of data and pull out numbers like 2 , 4 , 6 and so forth in the array without fully being aware of each and every step your brain is taking to solve it. If this is challenging, try using large sets of data as it will override your brain’s ability to naturally solve the problem just by looking at it. That helps you work through the real algorithm.

Let’s go through the first array [1]

1. Look at the only element in the array [1]
2. Decide if it is even. It is not
3. Notice that there are no more elements in this array
4. Determine there are no even numbers in this provided array
5. Return an empty array

Let’s go through the array [1, 2]

1. Look at the first element in array [1, 2]
2. It is 1
3. Decide if it is even. It is not
4. Look at the next element in the array
5. It is 2
6. Decide if it is even. It is even
7. Make an array evenNumbers and add 2 to this array
8. Notice that there are no more elements in this array
9. Return the array evenNumbers which is [2]

I go through this a few more times. Notice how the steps I wrote down for [1] varies slightly from [1, 2]. That is why I try to go through a couple of different sets. I have some sets with just one element, some with floats instead of just integers, some with multiple digits in an element, and some with negatives just to be safe.

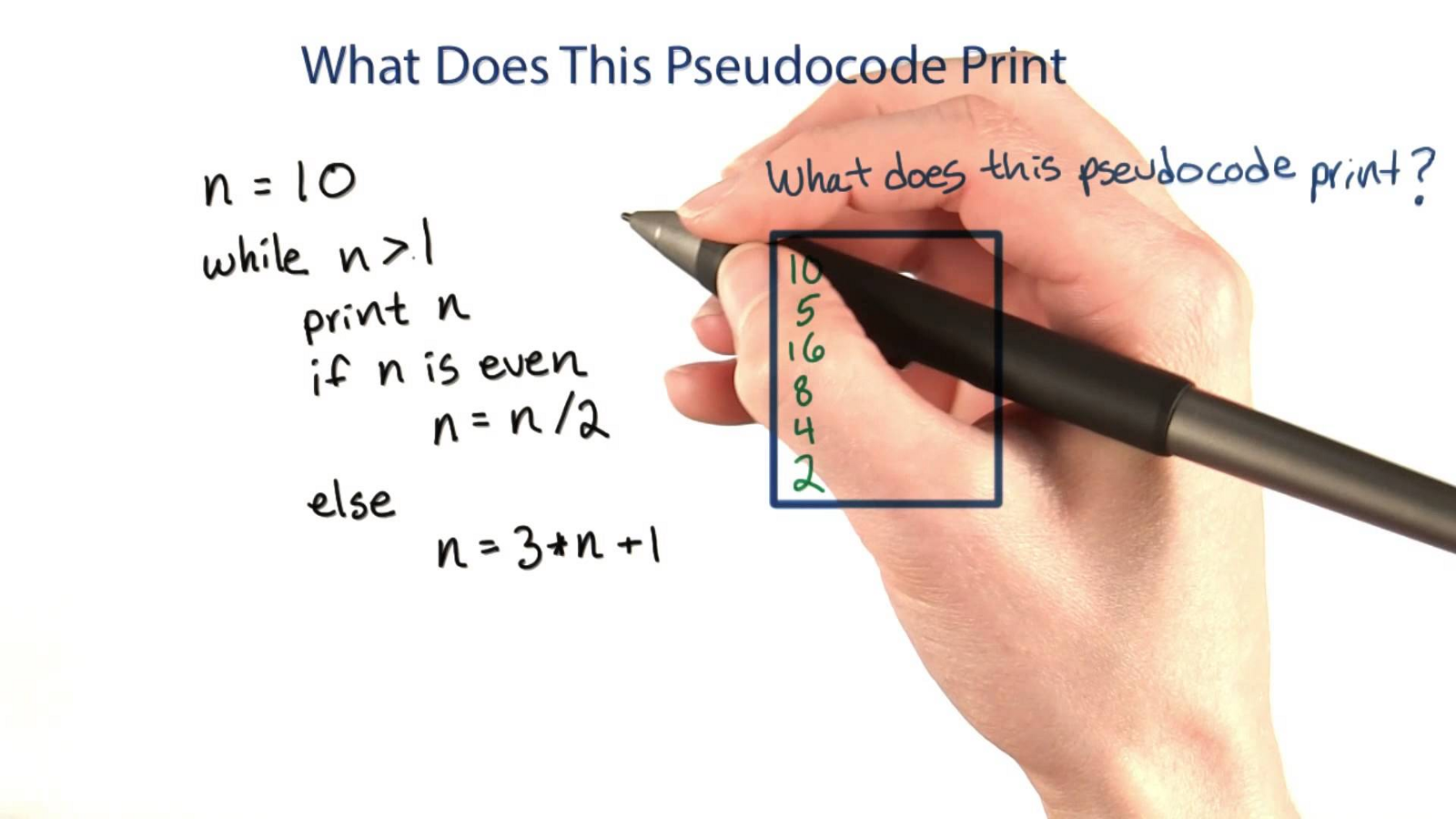
**3. Simplify and optimize your steps**

Look for patterns and see if there’s anything you can generalize. See if you can reduce any steps or if you are repeating any steps.

1. Create a function selectEvenNumbers
2. Create a new empty array evenNumbers where I store even numbers, if any
3. Go through each element in the array [1, 2]
4. Find the first element
5. Decide if it is even by seeing if it is divisible by 2. If it is even, I add that to evenNumbers
6. Find the next element
7. Repeat step #4
8. Repeat step #5 and #4 until there are no more elements in this array
9. Return the array evenNumbers, regardless of whether it has anything in it

This approach may remind you of [Mathematical Induction](https://en.wikipedia.org/wiki/Mathematical_induction) in that you:

1. Show it is true for n = 1, n = 2, ...
2. Suppose it is true for n = k
3. Prove it is true for n = k + 1



Example of pseudocode

**4. Write pseudocode**

Even after you’ve worked out general steps, writing out pseudocode that you can translate into code will help with defining the structure of your code and make coding a lot easier. **Write pseudocode line by line.** You can do this either on paper or as comments in your code editor. If you’re starting out and find blank screens to be daunting or distracting, I recommend doing it on paper.

Pseudocode generally does not actually have specific rules in particular but sometimes, I might end up including some syntax from a language just because I am familiar enough with an aspect of the programming language. **Don’t get caught up with the syntax. Focus on the logic and steps.**

For our problem, there are many different ways to do this. For example, you can use filter but for the sake of keeping this example as easy to follow along as possible, we will use a basic for loop for now (but we will use filter later when we refactor our code).

Here is an example of pseudocode that has more words:

function selectEvenNumbers

create an array evenNumbers and set that equal to an empty array

for each element in that array  
 see if that element is even  
 if element is even (if there is a remainder when divided by 2)  
 add to that to the array evenNumbers

return evenNumbers

Here is an example of pseudocode that has fewer words:

function selectEvenNumbers

evenNumbers = []

for i = 0 to i = length of evenNumbers  
 if (element % 2 === 0)   
 add to that to the array evenNumbers

return evenNumbers

Either way is fine as long as you are writing it out line-by-line and understand the logic on each line.

Refer back to the problem to make sure you are on track.

**5. Translate pseudocode into code and debug**

When you have your pseudocode ready, translate each line into real code in the language you are working on. We will use JavaScript for this example.

If you wrote it out on paper, type this up as comments in your code editor. Then replace each line in your pseudocode.

Then I call the function and give it some sample sets of data we used earlier. I use them to see if my code returns the results I want. You can also write tests to check if the actual output is equal to the expected output.

selectEvenNumbers([1])  
selectEvenNumbers([1, 2])  
selectEvenNumbers([1, 2, 3, 4, 5, 6])  
selectEvenNumbers([-200.25])  
selectEvenNumbers([-800.1, 2000, 3.1, -1000.25, 42, 600])

I generally use console.log() after each variable or line or so. This helps me **check if the values and code are behaving as expected before I move on**. By doing this, I catch any issues before I get too far. Below is an example of what values I would check when I am first starting out. I do this throughout my code as I type it out.

function selectEvenNumbers(arrayofNumbers) {

let evenNumbers = []  
 console.log(evenNumbers) // I remove this after checking output  
 console.log(arrayofNumbers) // I remove this after checking output

}

After working though each line of my pseudocode, below is what we end up with. // is what the line was in pseudocode. Text that is **bolded** is the actual code in JavaScript.

// function selectEvenNumbers  
**function selectEvenNumbers(arrayofNumbers) {**

// evenNumbers = []  
 **let evenNumbers = []**

// for i = 0 to i = length of evenNumbers  
 **for (var i = 0; i < arrayofNumbers.length; i++) {**

// if (element % 2 === 0)   
 **if (arrayofNumbers[i] % 2 === 0) {**

// add to that to the array evenNumbers  
 **evenNumbers.push(arrayofNumbers[i])**  
 **}**  
 **}**

// return evenNumbers  
 **return evenNumbers  
}**

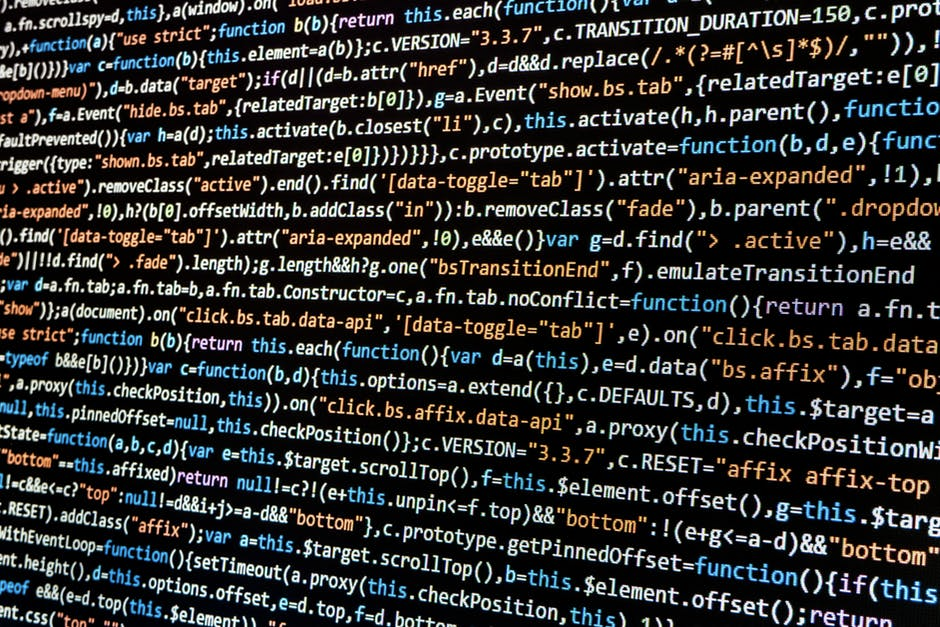
I get rid of the pseudocode to avoid confusion.

function selectEvenNumbers(arrayofNumbers) {  
 let evenNumbers = []

for (var i = 0; i < arrayofNumbers.length; i++) {  
 if (arrayofNumbers[i] % 2 === 0) {  
 evenNumbers.push(arrayofNumbers[i])  
 }  
 }

return evenNumbers  
}

Sometimes new developers will get hung up with the syntax that it becomes difficult to move forward. **Remember that syntax will come more naturally over time and there is no shame in referencing material for the correct syntax later on when coding.**



**6. Simplify and optimize your code**

You’ve probably noticed by now that simplifying and optimizing are recurring themes.

*“Simplicity is prerequisite for reliability.”*

*— Edsger W. Dijkstra, Dutch computer scientist and early pioneer in many research areas of computing science*

In this example, one way of optimizing it would be to filter out items from an array by returning a new array using filter. This way, we don’t have to define another variable evenNumbers because filter will return a new array with copies of elements that match the filter. This will not change the original array. We also don’t need to use a for loop with this approach. filter will go through each item, return either true, to have that element in the array, or false to skip it.

function selectEvenNumbers(arrayofNumbers) {  
 let evenNumbers = arrayofNumbers.filter(n => n % 2 === 0)  
 return evenNumbers  
}

Simplifying and optimizing your code may require you to iterate a few times, identifying ways to further simplify and optimize code.

Here are some questions to keep in mind:

* **What are your goals for simplifying and optimizing?** The goals will depend on your team’s style or your personal preference. Are you trying to condense the code as much as possible? Is the goal to make it the code more readable? If that’s the case, you may prefer taking that extra line to define the variable or compute something rather than trying to define and compute all in one line.
* **How else can you make the code more readable?**
* **Are there any more extra steps you can take out?**
* **Are there any variables or functions you ended up not even needing or using?**
* **Are you repeating some steps a lot?** See if you can define in another function.
* **Are there better ways to handle edge cases?**

*“Programs must be written for people to read, and only incidentally for machines to execute.”*

*— Gerald Jay Sussman and Hal Abelson, Authors of “Structure and Interpretation of Computer Programs”*

**7. Debug**

This step really should be throughout the process. Debugging throughout will help you catch any syntax errors or gaps in logic sooner rather than later. Take advantage of your Integrated Development Environment (IDE) and debugger. When I encounter bugs, I trace the code line-by-line to see if there was anything that did not go as expected. Here are some techniques I use:

* Check the console to see what the error message says. Sometimes it’ll point out a line number I need to check. This gives me a rough idea of where to start, although the issue sometimes may not be at this line at all.
* Comment out chunks or lines of code and output what I have so far to quickly see if the code is behaving how I expected. I can always uncomment the code as needed.
* Use other sample data if there are scenarios I did not think of and see if the code will still work.
* Save different versions of my file if I am trying out a completely different approach. I don’t want to lose any of my work if I end up wanting to revert back to it!

*“The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.”*

*— Brian W. Kernighan, Computer Science Professor at Princeton University*

**8. Write useful comments**

You may not always remember what every single line meant a month later. And someone else working on your code may not know either. That’s why it’s important to write useful comments to avoid problems and save time later on if you need to come back to it.

Stay away from comments such as:

// This is an array. Iterate through it.

// This is a variable

**I try to write brief, high-level comments that help me understand what’s going on if it is not obvious.** This comes in handy when I am working on more complex problems. It helps understand what a particular function is doing and why. Through the use of clear variable names, function names, and comments, you (and others) should be able to understand:

* What is this code for?
* What is it doing?



**9. Get feedback through code reviews**

Get feedback from your teammates, professors, and other developers. Check out [Stack Overflow](https://stackoverflow.com/). See how others tackled the problem and learn from them. There are sometimes several ways to approach a problem. Find out what they are and you’ll get better and quicker at coming up with them yourself.

*“No matter how slow you are writing clean code; you will always be slower if you make a mess.”*

*— Uncle Bob Martin, Software Engineer and Co-author of the Agile Manifesto*

**10. Practice, practice, practice**

Even experienced developers are always practicing and learning. If you get helpful feedback, implement it. Redo a problem or do similar problems. Keep pushing yourself. **With each problem you solve, the better a developer you become.**Celebrate each success and be sure to remember how far you’ve come. Remember that programming, like with anything, comes easier and more naturally with time.

*“Take pride in how far you’ve come. Have faith in how far you can go. But don’t forget to enjoy the journey.”*

*— Michael Josephson, Founder of Joseph and Edna Josephson Institute of Ethics*