## **Problem Solving**

### **Introduction**

Before we start digging into some pretty nifty JavaScript, we need to begin talking about *problem solving*: the most important skill a developer needs.

Problem solving is the core thing software developers do. The programming languages and tools they use are secondary to this fundamental skill.

V. Anton Spraul defines problem solving in programming as:

“Problem solving is writing an original program that performs a particular set of tasks and meets all stated constraints.” - Think Like a Programmer

The set of tasks can range from solving small coding exercises all the way up to building a social network site like Facebook or a search engine like Google. Each problem has its own set of constraints, for example, high performance and scalability may not matter too much in a coding exercise but it will be vital in apps like Google that need to service billions of search queries each day.

New programmers often find problem solving the hardest skill to build. It’s not uncommon for budding programmers to breeze through learning syntax and programming concepts, yet when trying to code something on their own, they find themselves staring blankly at their text editor not knowing where to start.

The best way to improve your problem solving ability is by building experience by making lots and lots of programs. The more practice you have the better you’ll be prepared to solve real world problems.

In this lesson we will walk through a few techniques that can be used to help with the problem solving process.

### **Learning Outcomes**

By the end of this lesson, you should be able to do the following:

* Explain the three steps in the problem solving process.
* Explain what pseudocode is and be able to use it to solve problems.
* Be able to break a problem down into subproblems.

### **Understand the Problem**

The first step to solving a problem is understanding exactly what the problem is. If you don’t understand the problem you won’t know when you’ve successfully solved it and may waste a lot of time on a wrong solution.

To gain clarity and understanding of the problem, write it down on paper, reword it in plain English until it makes sense to you, and draw diagrams if that helps. When you can explain the problem to someone else in plain English, you understand it.

### **Plan**

Now that you know what you’re aiming to solve, don’t jump into coding just yet. It’s time to plan out how you’re going to solve it first. Some of the questions you should answer at this stage of the process:

* Does your program have a user interface? What will it look like? What functionality will the interface have? Sketch this out on paper.
* What inputs will your program have? Will the user enter data or will you get input from somewhere else?
* What’s the desired output?
* Given your inputs, what are the steps necessary to return the desired output?

The last question is where you will write out an algorithm to solve the problem. You can think of an algorithm as a recipe for solving a particular problem. It defines the steps that need to be taken by the computer to solve a problem in pseudo code.

### **Pseudo Code**

Pseudo code is writing out the logic for your program in natural language instead of code. It helps you slow down and think through the steps your program will have to go through to solve the problem.

Here’s an example of what the pseudo code for a simple program that prints all numbers up to an inputted number might look like:

When the user inputs a number

Initialize a counter variable and set its value to zero

While counter is smaller than user inputted number increment the counter by one

Print the value of the counter variable

This is a very simple program to demonstrate how pseudo code looks. There will be more examples of pseudo code included in the assignments.

### **Divide and Conquer**

From your planning, you should have identified some subproblems of the big problem you’re solving. Each of the steps in the algorithm we wrote out in the last section are subproblems. Pick the smallest or simplest one and start there with coding.

It’s important to remember that you might not know all the steps that you might need up front, so your algorithm may be incomplete -— this is fine. Getting started with and solving one of the subproblems you have identified in the planning stage often reveals the next subproblem you can work on. Or, if you already know the next subproblem, it’s often simpler with the first subproblem solved.

Many beginners try to solve the big problem in one go. Don’t do this. If the problem is sufficiently complex, you’ll get yourself tied in knots and make life a lot harder for yourself. Decomposing problems into smaller and easier to solve subproblems is a much better approach. Decomposition is the main way to deal with complexity, making problems easier and more approachable to solve and understand.

In short, break the big problem down and solve each of the smaller problems until you’ve solved the big problem.

### **Solving Fizz Buzz**

To demonstrate this workflow in action, let’s solve a common programming exercise: [FizzBuzz](https://en.wikipedia.org/wiki/Fizz_buzz).

#### **Understanding The Problem**

Write a program that takes a user’s input and prints the numbers from one to the number the user entered. However, for multiples of three print Fizz instead of the number and for the multiples of five print Buzz. For numbers which are multiples of both three and five print FizzBuzz.

This is the big picture problem we will be solving. It’s pretty simple so we may not need to reword it. But we can always make it clearer by rewriting it.

Write a program that allows the user to enter a number, print each number between one and the number the user entered, but for numbers that divide by three without a remainder print Fizz instead. For numbers that divide by 5 without a remainder print Buzz and finally for numbers that divide by both three and five without a remainder print FizzBuzz.

#### **Plan**

Does your program have an interface? What will it look like? Our FizzBuzz solution will be a command line program, so we don’t need an interface. The only user interaction will be allowing users to enter a number.

What inputs will your program have? Will the user enter data or will you get input from somewhere else? The user will enter a number from the command line.

What’s the desired output? The desired output is a list of numbers from 1 to the number the user entered. But each number that is divisible by 3 will output Fizz, each number that is divisible by 5 will output Buzz and each number that is divisible by both 3 and 5 will output FizzBuzz.

Given your inputs, what are the steps necessary to return the desired output? The algorithm in pseudo code for this problem:

When a user inputs a number

Loop from 1 to the entered number

If the current number is divisible by 3 then print "Fizz"

If the current number is divisible by 5 then print "Buzz"

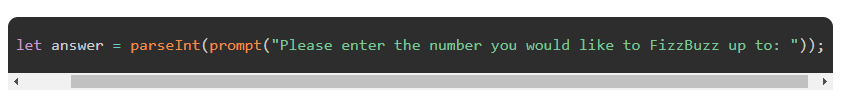
If the current number is divisible by 3 and 5 then print "FizzBuzz"

Otherwise print the current number

#### **Divide and Conquer (Implement)**

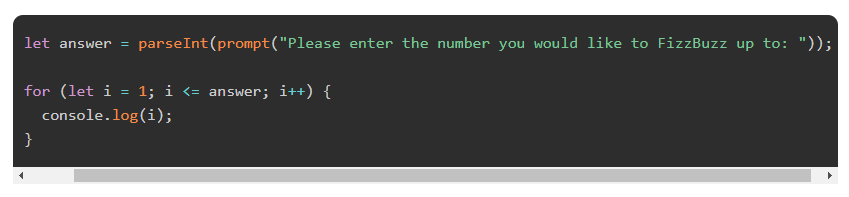
As we can see from the algorithm we developed, the first subproblem we can solve is getting input from the user. So let’s start there and verify it works by printing the entered number.

With JavaScript, we’ll use the “prompt” method.



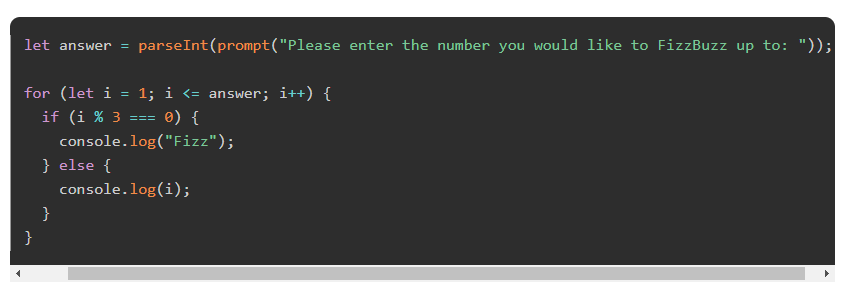
The above code should create a little popup box that asks the user for a number. The input we get back will be stored in our variable answer. Note: We wrapped the prompt call in a parseInt function so that a number is returned from the user’s input.

With that done, let’s move on to the next subproblem: “Loop from 1 to the entered number”. There are many ways to do this in JavaScript. One of the common ways - that you actually see in many other languages like Java, C++, and Ruby - is with the for-loop:



If you haven’t seen this before and it looks strange, it’s actually straightforward. We declare a variable i and assign it 1: the initial value of the variable i in our loop (NOTE: Most of the time, programmers find themselves looping from 0. Due to the needs of our program, we’re starting from 1). The second clause, i <= answer is our condition. We want to loop until i is greater than answer. The third clause, i++, tells our loop to increment i by 1 every iteration. As a result, if the user inputs 10, this loop would print numbers 1 - 10 to the console.

With that working, let’s move on to the next problem: If the current number is divisible by 3, then print Fizz.

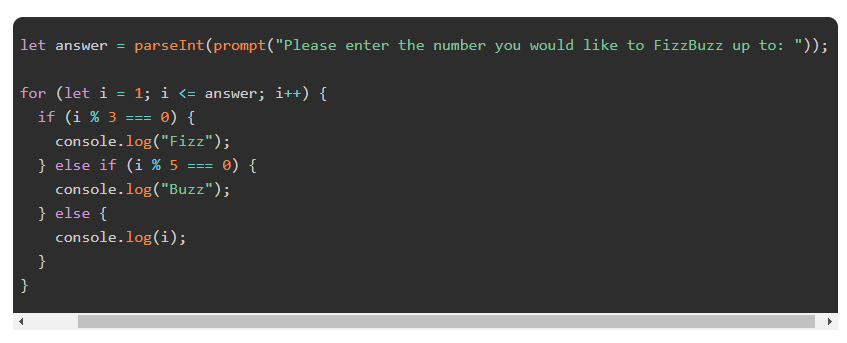


We are using the modulus operator (%) here to divide the current number by three. If you recall from a previous lesson, the modulus operator returns the remainder of a division. So if a remainder of 0 is returned from the division, it means the current number is divisible by 3.

After this change the program will now output this when you run it and the user inputs 10:



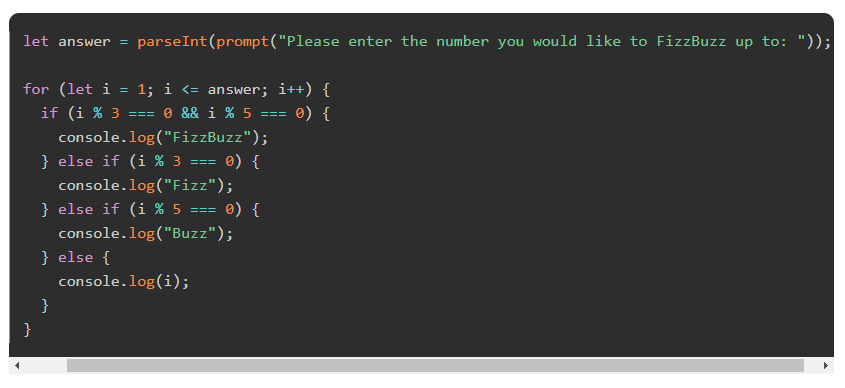
The program is starting to take shape. The final few subproblems should be easy to solve as the basic structure is in place and they are just different variations of the condition we’ve already got in place. Let’s tackle the next one: If the current number is divisible by 5 then print Buzz.



When you run the program now, you should see this output if the user inputs 10:



We have one more subproblem to solve to complete the program: If the current number is divisible by 3 and 5 then print FizzBuzz.



We’ve had to move the conditionals around a little to get it to work. The first condition now checks if i is divisible by 3 and 5 instead of checking if i is just divisible by 3. We’ve had to do this because if we kept it the way it was, it would run the first condition if (i % 3 === 0), so that if i was divisible by 3, it would print Fizz and then move on to the next number in the iteration, even if i was divisible by 5 as well.

With the condition if (i % 3 === 0 && i % 5 === 0) coming first, we check that i is divisible by both 3 and 5 before moving on to check if it is divisible by 3 or 5 individually in the else if conditions.

The program is now complete! If you run it now you should get this output when the user inputs 20:



### **Assignment**

1. Read [How to Think Like a Programmer - Lessons in Problem Solving](https://www.freecodecamp.org/news/how-to-think-like-a-programmer-lessons-in-problem-solving-d1d8bf1de7d2/) by Richard Reis.
2. Watch [How to Begin Thinking Like a Programmer](https://www.youtube.com/watch?v=azcrPFhaY9k) by Coding Tech. It’s an hour long but packed full of information and definitely worth your time watching.
3. Read this [What is Pseudo Coding](https://www.vikingcodeschool.com/software-engineering-basics/what-is-pseudo-coding) article from The Viking Code School.

### **Additional Resources**

This section contains helpful links to other content. It isn’t required, so consider it supplemental.

* Read [Think Like a Programmer: An Introduction to Creative Problem Solving](https://www.amazon.com/Think-Like-Programmer-Introduction-Creative/dp/1593274246/ref=sr_1_3?ie=UTF8&qid=1540326000&sr=8-3&keywords=think+like+a+programmer) (*not free*). This book’s examples are in C++, but you will understand everything since the main idea of the book is to teach programmers to better solve problems. It’s an amazing book and worth every penny. It will make you a better programmer.
* Watch this [video on repetitive programming techniques](https://ocw.mit.edu/resources/res-tll-004-stem-concept-videos-fall-2013/videos/problem-solving/basic-programming-techniques/).

### **Knowledge Check**

This section contains questions for you to check your understanding of this lesson. If you’re having trouble answering the questions below on your own, review the material above to find the answer.

* [What are the three stages in the problem solving process?](https://www.theodinproject.com/paths/foundations/courses/foundations/lessons/problem-solving#problem-solving-stages)
* [Why is it important to clearly understand the problem first?](https://www.theodinproject.com/paths/foundations/courses/foundations/lessons/problem-solving#important-understand-problem)
* [What can you do to help get a clearer understanding of the problem?](https://www.theodinproject.com/paths/foundations/courses/foundations/lessons/problem-solving#help-understand-problem)
* [What are some of the things you should do in the planning stage of the problem solving process?](https://www.theodinproject.com/paths/foundations/courses/foundations/lessons/problem-solving#planning-stage)
* [What is an algorithm?](https://www.theodinproject.com/paths/foundations/courses/foundations/lessons/problem-solving#algorithm)
* [What is pseudo code?](https://www.theodinproject.com/paths/foundations/courses/foundations/lessons/problem-solving#pseudo)
* [What are the advantages of breaking a problem down and solving the smaller problems?](https://www.theodinproject.com/paths/foundations/courses/foundations/lessons/problem-solving#breaking-problem)