Algorithm Challenges

The Dojo Collection

Version 1.1.7
May 8, 2017

By Martin Puryear

CODING
DOJO

First Printing and eBook Release: 2016

Coding Dojo, Inc.
10777 Main Street, Suite 100
Bellevue, WA 98004

www.codingdojo.com

ISDN: 978-1-365-45721-0

Algorithms Course Summary

Every day for **at least one hour**, we will challenge you with algorithm problems from our sequential curriculum.

Daily Operation
Each onsite cohort will divide into groups, and groups will be tasked with solving algorithms on whiteboards. In the session's last 15 minutes, one group will present solutions, as will your instructor/TA. Online, challenges will be posted by the instructor at a common location.

Goals
One important goal of the course is to get you comfortable describing your code's functionality. This is important for technical interviews, where you are asked to demonstrate your knowledge using only whiteboards.

Another important goal is to familiarize yourself with algorithms and data structures that solve complex problems efficiently, even as they scale worldwide.

Rules
1. Show up – the only way to rewire your brain to think like a computer is *repetition*. Make sure you're on time for every session to get the challenge's introduction.

2. No Laptops – to simulate a technical interview, do not use laptop or refer to old code during challenges. Until directed otherwise, work problems on a whiteboard.

3. Be respectful – being able to walk others through your algorithm, explaining how it works, is as important as correctness. There will be many chances to discuss with peers or present to the class. All students should give the speaker full attention and respect. Public speaking is a common fear; we will learn to conquer our nerves. This only happens in a welcoming environment.

4. Work in groups – group work requires you to articulate your thoughts and describe your code, skills that are also useful when working in engineering teams more generally. Groups that are too large do not lead to participation by all group members. *Don't be just a spectator!* Unless otherwise directed, **solve algorithm challenges in groups of 2 or 3** – no smaller or larger.

Presenting
Every day, two groups present solutions. Make sure all group members have a chance to describe the algorithm; give presenting groups the proper respect.

Questions to ask about solutions
1. Is it clear and understandable?
Can you easily explain functionality and lead the listener through a T-diagram? Does the code self-describe effectively, or do you find yourself having to explain the meaning of the variables 'x' and 'i'?

2. Is the output correct?
Does your algorithm produce the required results? Is your algorithm resilient in the face of unexpected inputs or even intentional attempts to get it to crash?

3. Is it concise?
Remember the acronym DRY (Don't Repeat Yourself). Less code is better, so long as it is fully understandable. Pull any duplicate code into helper functions.

4. Is it efficient?
Does your function contain only necessary statements, and does it require only necessary memory? Does it stay efficient (in run time as well as memory usage), as the input size gets very large? Are you mindful of any intentional tradeoffs of time vs. space (improving run time by using more memory, or vice versa)?

Tips
- Think out loud, to provide a window to your thinking. You may even get help if you are on the wrong track!
- Describe assumptions. Clarify before writing code.
- List sample inputs, along with expected outputs. This validates your understanding of the problem.
- Don't bog down. Add a comment, then move on.
- Break big problems down into smaller problems.
- Focus on correct outputs, *then* on *'correct'* solutions.
- Don't stress! Algorithm challenges are brain cardio. This is not an evaluation of your abilities or expertise!
- Have fun! *Problem analysis* is a fundamental skill that makes you a more effective software engineer.

Table of Contents

Chapter 0 – Foundation Concepts

Computers, Software, and Source Code

Computers are amazing. They rapidly perform complex calculations, store immense amounts of data, and almost instantly retrieve specific bits of information from mountains of data. In addition to being fast, they appear superhuman in their ability to use information to make decisions. How do they do it?

Simply put, computers are machines. We as humans are skilled at creating tools that perform *specific* tasks *very* well: a toaster, for example, or a lawn mower. Computers are tools built from components such as semiconductor *chips*, and ongoing advances in material science enable these pieces to get smaller and faster with every successive year (see *Moore's Law*). This is why computers have become so breathtakingly fast, but it only partially explains why they can seem so *smart*. It doesn't really tell us why they are so universally relied upon to solve such a diverse range of problems across today's world.

Computing is exciting because computing devices are flexible – they can be *taught* to do things not imagined when they were originally built. Science fiction may become science in the future, but for today they only know what we *tell* them; they only do as they are told. Who *teaches* them, who *tells* them what to do? Software engineers, developers, programmers! You are reading this, so you probably intend to be one too. From a computer's viewpoint, you are training to become an educator. (-:

How do programmers tell computers what to do? We create ***Software***. Software is a sequence of instructions that we build and provide to a computer, which then "mindlessly" runs those instructions. Computers cannot natively understand human language, nor can humans read the language of semiconductor components. To talk with computers, we need a "go-between" format: something that software engineers can understand, yet can also be translated into machine language instructions.

Many of these "go-between" languages have been created: PHP, Python, Ruby, JavaScript, Swift, C#, Java, Perl, Erlang, Go, Rust, and others – even HTML and CSS! Each language has differing strengths and therefore is useful in different situations. Programming languages all do essentially the same things though: they read in a series of human-readable steps instructing a computer how to respond, and they translate the steps into a format the computer can understand and later execute.

All these sequences of instructions, written in programming languages like JavaScript, are what we call ***Source Code****.* How and when this source code is translated into *machine code* will depend on the language and the machine. *Interpreted languages* like PHP, Python and Ruby translate from source code into machine code "on the fly", immediately before a computer needs it. *Compiled languages* do some or all of this translation ahead of time. As we said earlier, a computer simply follows instructions it was given. More specifically, though, it executes (*runs*) machine code that was built (*translated*) from some piece of source code: code that was written by a software engineer.

Let's teach you how to think like a computer, so you can write effective source code. We have chosen to use JavaScript as the programming language for this book, so along the way you'll learn the specifics of that language. With very few exceptions, however, these concepts are universal.

Chapter 0 – Foundation Concepts

Code Flow

When a computer executes (runs) a piece of code, it simply reads each line from the beginning of the file, executing it in order. When the computer gets to the end of the lines to execute, it is finished with that program. There may be *other* programs running on that computer at the same time (e.g. pieces of the operating system that update the monitor screen), but as far as your program goes, when the computer's execution gets to the end of the source you've given it, the program is done and the computer has completed running your code.

It isn't necessary for your code to be purely linear from the top of the file to the end of the file, however. You can instruct the computer to execute the same section of code multiple times – this is called a program **`LOOP`**. Also, you can have the computer jump to a different section of your code, based on whether a certain condition is true or false – this is called an **`IF-ELSE`** statement (or a conditional). Finally, for code that you expect to use often, whether called by various places in your own code, or perhaps even called by others' code, you can separate this out and give it a specific label so that it can be called directly. This is called a **`FUNCTION`**. More on each of these later.

Variables

Imagine if you had two objects (a book and a ball) that you wanted to carry around in your hands. With two hands, it is easy enough to carry two objects. However, what if you also had a sandwich? You don't have enough hands, so you need one or more containers for each of the objects. What if you had a box with a label on it, inside which you can put one of your objects. The box is closed, so all you see is the label, but it is easy enough to open the box and look inside. This is essentially what a *variable* does.

A variable is a specific spot in memory, with a label that you give it. You can put anything you want into that memory location and later refer to the value of that memory, by using the label. The statement:

```
var myName = 'Martin';
```

creates a variable, gives it a label of **`myName`**, and puts a value of **`"Martin"`** into that memory location. Later, to reference or inspect the value that you stored there, you simply refer to the label **`myName`**. For example (jumping ahead to the upcoming Printing to the Console topic), to *display* the value in the variable **`myName`**, do this:

```
console.log(myName);
```

Chapter 0 – Foundation Concepts

Data Types

Containers exist to hold things. You would create a variable because you want it to store some value – some piece of information. A value could be a number, or a sentence made of text characters, or something else. Specifically, JavaScript has a few *data types*, and all values are one of those types. Of these six data types, three are important to mention right now. These are Number, String, and Boolean.

In JavaScript, a *Number* can store a huge range of numerical values from extremely large values to microscopically small ones, to incredibly negative values as well. If you know other programming languages, you might be accustomed to making a distinction between integers and floating-point numbers – JavaScript makes no such distinction.

A *String* is any sequence of characters, contained between quotation marks. In JavaScript, you can use either single-quotes or double-quotes. Either way, just make sure to close the string the same way you opened it. **`'Word'`** and **`"wurd"`** are both fine, but **`'weird"`** and **`"whoa!'`** are not.

Finally, a *Boolean* has only two possible values: **`true`** and **`false`**. You can think of a Boolean like a traditional light switch, or perhaps a yes/no question on a test. Just as a light switch can be either *on* or *off*, and just as a yes/no question can be answered with either *yes* or *no*, likewise a Boolean must have a value of either **`true`** or **`false`** – there is nothing in-between.

One of the main things we do with variables, once they contain information of a certain data type, is to compare them. This is our next section.

Chapter 0 – Foundation Concepts

Not All Equals Signs Are the Same!

In many programming languages, you see both **=** and **==**. These mean different things! Code **`(X = Y)`** can be described as *"Set the value of X* to become *the value of Y"*, and you can describe **`(X == Y)`** as *"Is the value of X equivalent to the value of Y?"* It is more common – but less helpful right now while learning these concepts – verbalize these as *"Assign Y to X"* and *"Are X and Y equal?"*, respectively.

Use **=** to ***set*** *things*, **==** to ***test*** *things*. *Single-equals is for* assignment; *double-equals is for* comparison.

Many programming languages are extremely picky about data types. When asked to combine two values that have differing data types, some languages will halt with an error rather than do so. JavaScript, however, is not so strict. In fact, you could say that JavaScript is very *loosely* typed: it very willingly changes a variable's data type, whenever needed. Remember the == operator that we described previously? It *actually* means "*after converting X and Y to the same data type*, are their values equivalent?" If you want strict comparison without converting data types, use the **===** operator.

Generally, === is advised, unless you explicitly intend to equate values of differing types, such as **`{1,true,"1"}`** or **`{0,false,"0"}`**, etc. (If this last sentence doesn't make sense yet, don't worry.)

Quick quiz:

1. How many inputs are accepted by the == operator and the === operator, respectively?
2. Do inputs to == and === operators need to be the same data type, for the operators to function?
3. What is the data type of the output value produced by the == and === operators?

Hey! Don't just read on: *jot down answers* for those questions before moving on.

Done? OK, good….

Answers:

1. The == and === operators both accept *two* values (one before the operator, and one after).
2. *No*, two values need not be the same type for these operators. However, if they are not, === always returns **`false`**. The == internally converts values to the same type before comparing.
3. The == and === operators both return a *Boolean* value (**`true`** or **`false`**).

Now that you know just a little about Numbers, Strings and Booleans, let's start using them.

Chapter 0 – Foundation Concepts

Printing to the Console

Eventually, you will create fabulous web systems and/or applications that do very fancy things with graphical user interface. However, when we are first learning how to program, we start by having our programs write simple text messages (strings!) to the screen. In fact, the very first program that most people write in a new language is relatively well known as the “Hello World” program. In JavaScript, we can quickly send a text string to the *developer console*, which is where errors, warnings and other messages about our program go as well. This is not something that a normal user would ever look at, but it is the easiest way for us to print variables or other messages.

To log a message to the console, we use **`console.log()`**. Within the parentheses of this call, we put any message we want displayed. The **`console.log`** function always takes in a string. If we send it something that isn’t a string, JavaScript will first convert it to a string that it can print. It’s very obliging that way. So, our message to be logged could be a literal value (**`42`** or **`"Hello"`**), or a variable like this:

```
console.log("Hello World!");

var message = "Welcome to the Dojo";
console.log(message);
```

We can also combine literal strings and variables into a larger string for **`console.log`**, simply by adding them together. If you had a variable **`numDays`**, for example, you could log a message like this:

```
var numDays = 40;
console.log("It rained for " + numDays + " days and nights!");
```

Notice that we put a space at the end of our **`"It rained for "`** string, so that the console would log **`"It rained for 40 days and nights!"`** instead of **`"It rained for40days and nights!"`**

So, what would the following code print?

```
var greeting = "howdy";
console.log("greeting" + greeting);
```

It would print **`"greetinghowdy"`**, since we ask it to combine the literal string **`"greeting"`** with the value of the variable **`greeting`**, which is **`"howdy"`**. Make sense?

One last side note: if you *really insist* upon writing a string to the actual web page, you could use the old-fashioned function **`document.write()`**, such as **`document.write("Day #" + numDays)`**. However, you won’t use this function when creating real web pages, so why get into that habit now?

Chapter 0 – Foundation Concepts

Functions

Let's say that you are writing a piece of code that has five different places where it needs to print your name. As mentioned above, for code that you expect to call often, separate this out into a different part of your file, so these lines of code don't need to be duplicated each time you print your name. This is called a **`FUNCTION`**. Creating (or *declaring*) a function could look like this:

```
function sayMyName( )
{
   console.log("My name is Martin");
}
```

By using the special **`function`** word, you tell JavaScript that what follows is a set of source code that can be called at any time by simply referring to the **`sayMyName`** label. Note: *the code above does not actually call the function immediately*; it sets the function up for other code to use (call) it later.

'Calling' the *function* is also referred to as 'running' or 'executing' the function. If the above is how you declare a function, then the below is how you actually *run* that function:

```
sayMyName();
```

That's it! All you need to do is call that label, followed by open and close parentheses. The parentheses are what tell the computer to execute the function with that label, so don't forget those.

One last thing: there is nothing stopping a function from calling other functions (or in certain special situations, even calling itself!). You can see above that the **`sayMyName`** function does, in fact, call the built-in **`console.log`** function. Naturally, you would not expect **`console.log`** to run *until* you actually called it; in the same way, any code that you write will only start running when some other code calls it (maybe part of the computer browser or the operating system). So, except for the very first piece of code that runs when a hardware device starts up, all other code runs only because other code called it.

To review: when we *declare* a function, it allows some other *caller* to execute our function from some other place in the code, at some other time. It does not run the function immediately. So, if your source code file contains this:

```
function sayMyName( )
{
   console.log("My name is Martin");
}
```

…and then you execute that source code file, nothing would actually appear in the developer console. This is because no code ever called **`sayMyName()`**. You set it up, but never used it.

Chapter 0 – Foundation Concepts

Conditionals

If you are driving and reach a fork in the road, you must decide which way to go. Most likely, you will decide based on some very good reason. In code, there is a similar mechanism. **`IF`** statements look at the value of a variable, or perhaps compare two variables, and then execute certain lines of code if the result is what you expect. If you wish, you can also execute *other* lines of code if the result goes the other way. The important point is that each decision has only two possible outcomes. You have a certain test or comparison to be done; **`IF`** the test passes, **`THEN`** you execute certain code. If you wish, you can execute other code in the "*test did not pass*" (**`ELSE`**) case. This code would look like this:

```
if (myName == "Martin")
{
   console.log("Hey there Martin, how's it going?");
}
```

The **`IF`** statement is followed by parentheses that contain our *test*. Remember that we need to use a double-equals to create a comparison, and here we compare the value of variable **`myName`** to the string **`"Martin"`**. If the comparison passes, then we execute the next section of code within the curly braces. Otherwise, we skip that code. What if we want to greet users with a cheerful comment, even if they are not Martin? You can execute other code in this case (called the **`ELSE`**). We might write code like this:

```
if (myName == "Martin")
{
   console.log("Hey there Martin, how’s it going?");
}
else
{
   console.log("Greetings Earthling. Have a great day!");
}
```

Note: the "test" between the **`IF`**'s parentheses is an *expression* that results in a *Boolean* (a **`true`** or **`false`** value). This expression is evaluated when execution reaches the **`IF`**. If it evaluates to **`true`**, then your code enters the **`IF`** statement; if it evaluates to **`false`**, you enter the **`ELSE`** (if there is one).

Here is a brief code-based definition of how IF and ELSE statements operate:

```
if (EXPRESSION)    // EXPRESSION is evaluated upon reaching this line
{
   // body of 'IF': code runs only if EXPRESSION evaluates to true
}
else
{
   // body of 'ELSE': code runs only if EXPRESSION evaluates to false
}
```

Complex Conditionals

The expression evaluated by an `IF` statement can be more than a simple comparison. It can perform multiple comparisons, combining or modifying these values with AND, OR and NOT connectors – as long as it eventually produces a Boolean that the `IF` can use to determine *which way to go at the 'Y'*.

In spoken language, we can create compound conditional statements such as *"If it is Friday and I'm in a good mood, then let's go out and have some fun!"* Hence we would not go out if either it is *not* Friday, or if I'm *not* in a good mood. There are symbols in JavaScript that represent these logical AND, OR and NOT concepts.

The **AND** operator combines two logical tests, requiring **both** inputs to be `true` for the result to be `true`. The symbol for logical AND is a double-**&**, located between the two logical conditions, like this:

```
if (today == "Friday" && moodLevel >= 100)
{
   goDancing();
}
```

The OR is just the flip side of the AND. As conveyed above, we need both expressions to be `true`, but this could also be accurately described by the converse statements – such as *"If it is not Friday, or if I'm not in a good mood, then let's not go out."* The **OR** operator combines two logical tests, evaluating to `true` if **either** input is `true`. Another example might be "*If it is raining or if it is too far to walk, then let's call Uber instead!*" The logical OR is double-**|**, located between two logical conditions, like this:

```
if (raining == true || distanceMiles > 3)
{
   callUber();
}
```

We run the `callUber()` function unless *both* tests are *not* true (i.e. not raining *and* distance is three or less). So, we have the AND operator and the OR operator. The **NOT** operator inverts a single boolean: what would have been `true` becomes `false`, and what would have been `false` becomes `true`. Logical NOT is an exclamation point **!**. "If it isn't snowing, I'll wear shorts" could become this code:

```
if (!snowing)
{
   bravelyDonSomeShorts();
}
```

The function would be called only when we enter the `IF` statement, which is when `!snowing` is equal to `true`, which (rephrased) is when `snowing` is equal to `false`. Make sense?

Chaining and Nesting

So, we see that we can build complex expressions for a single IF statement. Why stop there? We can chain **`IF..ELSE`** statements with other **`IF..ELSE`** statements, like this:

```
if (myName == "Martin")
{
   console.log("Hey there Martin, how’s it going?");
}
else if (myName == "Beth")
{
   console.log("You look fabulous today!");
}
else
{
   console.log("Greetings Earthling. Have a great day!");
}
```

We can also nest an **`IF`** statement within another (or within an **`ELSE`**). This chaining and nesting can go on indefinitely, as needed. Can you decipher the below? When would you walk/fly/swim?

```
if (weather != "rainy")
{
   if (distanceToStadium < 3)
   {
      console.log("I think I’ll walk to the game.");
   }
   else
   {
      console.log("It’s a bit far, so maybe I’ll fly.");
   }
}
else
{
   console.log("Hey, I’m a duck! A little water is OK. I’ll swim.");
}
```

If not rainy, and if distance to stadium is less than 3 (miles?), then we walk. If not rainy, and if distance is 3 or more, then we fly. If weather is rainy (regardless of distance), then we swim. What an odd duck!

Chapter 0 – Foundation Concepts

Loops

Sometimes you will have lines of code that you want to run more than once in succession. It would be very wasteful to simply copy-and-paste that code over and over. Plus, if you ever needed to change the code, you would need to change all those lines one by one. What a mess! Instead, you can indicate that a section of code should be executed some number of times. Consider the following: "Do the next thing I tell you *four* times: hop on one foot." That would be much better than "Hop on one foot. Hop on one foot. Hop on one foot. Hop on one foot." (even though it is just as silly) Programming languages have the concept of a *LOOP* that is essentially a section of code that will be executed a certain number of times. There are a few different types of loops. The most common are **FOR** and **WHILE** *loops*. Shall we explore each of them in turn? Yes, let's.

FOR Loops

FOR *loops* are useful when you know how many times those lines of code will run. **WHILE** *loops* are slightly better when you don't know how many times to loop, but you will loop *while* a certain test continues to be true. To create a **FOR** *loop*, in addition to the code to be looped, you specify three things within parentheses following the **FOR**: any initial setup, a test that must be true in order to start the loop, and any code to be run at end of each time through the loop. Here is an annotated example:

```
//      A ;         B ;         D
for (var num = 1; num < 6; num = num + 1)
{
   // C
   console.log("I'm counting! The number is ", num);
}
// E
console.log("We are done. Goodbye world!");
```

The above will execute in this sequence: A - B-C-D - B-C-D - B-C-D - B-C-D - B-C-D - B - E.

Let's walk through this **FOR** loop in detail. Up front, local variable **num** is created and set to a value of 1. This *step A* happens exactly once, then we start looping. *Step B*: we compare **num** to 6. If it is less than 6, then the code within curly braces (*step C*) is executed, and then 1 is added to **num** (*step D*). We then return to *step B*. When the test at *step B* fails, we immediately exit without executing *step C* or *step D*. At that point, execution continues from *step E*, following our closed-curly-brace.

Said another way, we INIT, then we [TEST? – BODY – INCREMENT] while TEST is true, then we exit.

```
for (INITIALIZATION; TEST; INCREMENT/DECREMENT)
{
   // BODY of the loop –
   // this runs repeatedly as long as TEST is true
}
```

WHILE Loops

`WHILE` *loops* are similar to **`FOR`** *loops*, except with two pieces missing. First, there is no upfront setup like is built into a **`FOR`** loop. Also, unlike a **`FOR`** statement, a **`WHILE`** doesn't automatically include code that is executed at the end of each loop (our D above). **`WHILE`** *loops* are great when you don't know how many times (iterations) you will loop. Any **`FOR`** *loop* can be written as a **`WHILE`** *loop*. For example, the above **`FOR`** *loop* could be written instead as this **`WHILE`** *loop*, which would execute identically:

```
// A
var num = 1;
// B
while (num < 6)
{
   // C
   console.log("I'm counting! The number is " + num);
   // D
   num = num + 1;
}
// E
console.log("We are done. Goodbye world!");
```

Behaving identically with the above **`FOR`** loop, the **`WHILE`** code written immediately above will execute in this sequence: A - B-C-D - B-C-D - B-C-D - B-C-D - B-C-D - B - E.

Let's review before we move on. Anything we do with a **`FOR`** loop, we could achieve with a **`WHILE`** loop instead – and vice versa. So, when should we use **`FOR`** loops, and when should we use **`WHILE`** loops? Generally, use **`FOR`** loops when you know *exactly* how long a loop should run. Use **`WHILE`** loops when you have a condition that keeps the loop running (or that will cause the loop to stop), but you aren't sure exactly how many iterations that will require.

Other Loop Tips

Some developers like to increment a variable's value by running `num += 1;` this is the same as typing `num = num + 1`. You may sometimes see `num++` or even `++num`; both are equivalent to `+=`.

```
var index = 2;
index = index + 1;
index++;
// index now holds a value of 4
```

By that same token, we can decrement the value of num by running simply `num--;` or `--num;`. This is exactly the same as running `num = num – 1;` or `num -= 1.` There are `*=` and `/=` operators as well, that multiply and divide a number as you might expect.

```
var counter = 5;
counter = counter - 1;   // counter now holds a value of 4
counter--;               // counter is now 3
counter *= 6;            // counter is 18
counter /= 2;            // counter == 9
```

Furthermore, not every loop must increment by one. Can you guess what the following would output?

```
for (var num = 10; num > 2; num = num - 1)
{
   console.log('num is currently', num);
}
```

Yes, that's right: this `FOR` loop counts backwards by ones, starting at 10, while num is greater than 2. So, it would count `10,9,8,7,6,5,4,3`.

How would you print all *even* numbers from 1 to 1000000? How would you print all the *multiples of 7* (7, 14, ...) up to 100? Understanding how to use `FOR` loops is critical, so get really familiar with this.

Loops and Code Flow

With more complex loops, you might need to break out of a loop early, or to skip the current pass but continue looping. In JavaScript, you can use special **`BREAK`** and **`CONTINUE`** keywords to do this. The **`break`** keyword *immediately exits the specific loop you are currently in* and continues immediately following the loop. Even the final end-loop statement (**`num = num + 1`** above) will not be executed. A **`continue`** *skips the rest of the current pass* through the loop, but any loop-end statement *is* executed and looping will continue. With both, once they run, any subsequent code within the loop is skipped.

Here's an example. The following code prints the first two lines, but then immediately exits the loop.

```
var num = 1;
while (num < 5)
{
   if (num == 3)
   {
      break;
      // if you have code here, it will never run!
   }
   console.log("I'm counting! The number is ", num);
   num = num + 1;   // if we break, these lines won't run
}
```

I'm counting! The number is 1
I'm counting! The number is 2

The below counts from 1 to 4, printing something about each number, but completely forgets about 3, because when num == 3, a **`continue`** skips the rest of the loop and proceeds (after adding 1 to *num*).

```
for (var num = 1; num < 5; num += 1)
{
   if (num == 3)
   {
      continue;
      // if you have additional code down here, it will never run!
   }
   console.log("I'm counting! The number is ", num);
}
```

I'm counting! The number is 1
I'm counting! The number is 2
I'm counting! The number is 4

Loops commonly use **`break`**. Get comfortable using it to loop for a number of iterations, then exit when you encounter a certain condition. With this, it isn't preposterous to see **`while(true)`**! My goodness.

Parameters

Being able to call another function can be helpful for eliminating a lot of duplicate source code. That said, a function that always does exactly the same thing will be useful only in specific situations. It would be better if functions were more flexible and could be customized in some way. Fortunately, you can pass values into functions, so that the functions can behave differently depending on those values. The caller simply inserts these values (called *arguments*) between the parentheses, when it executes the function. When the function is executed, those values are copied in and are available like any other variable. Specifically, inside the function, these copied-in values are referred to as *parameters*.

For example, let's say that we have pulled our friendly greeting code above into a separate function, named **`greetSomeone`**. This function could include a parameter that is used by the code inside to customize the greeting, just as we did in our standalone code above. Depending on the argument that the caller sends in, our function would have different outcomes. Tying together the ideas of functions, parameters, conditionals and printing, this code could look like this:

```
function greetSomeone(person)
{
   if (person == "Martin") {
      console.log("Yo dawg, howz it goin?");
   }
   else
   {
      console.log("Greetings Earthling!");
   }
}
```

You might notice in the code above that there are curly braces that are not alone on their own lines, as they were in the previous code examples. The JavaScript language does not care whether you give these their own line or include them at the end of the previous line, as long as they are present. Really, braces are a way to indicate to the system some number of lines of code that it should treat as a single group. Without these, **`IF..ELSE`** and **`WHILE`** and **`FOR`** statements will only operate on a single line of code. Even if your loop is only a single line of code (and hence would work without braces), it is always safer to include these, in case you add more code to your loop later – and to reinforce good habits.

So, is it better to include these on their own lines, or to append them to the ends of the previous lines? This is really a matter of choice: *as long as you include them* (and you always should), JavaScript doesn't care about a few extra new-line characters or extra spaces. Over time you will develop your own ***coding style***, writing source code in the way that is most understandable to you. Keep in mind though that when you join a software team, you will likely need to adopt the team's coding style (if they have one). If they don't, then even in that case you should generally match the style of existing code in the source files where you are working. So, as you develop your personal style, it is best to stay flexible.

Chapter 0 – Foundation Concepts

Foundations Review

Hopefully, this first chapter made you more comfortable with the essential building blocks of software. Below is a summary of the ideas we covered.

Computers can do amazing things but they need to be told what to do. We tell them what to do by running software. Software is generally built from *source code*, which is readable by humans, and is a sequence of basic steps that a computer will follow exactly. There are many different software languages (such as JavaScript), with different ways of expressing these basic steps, however most of the main concepts are universal. Our job is to break down problems into these steps, and then the computer will *run* that code when told. Generally, when source code is run, it executes from the first line linearly to the last line. However, we can change this flow by adding "fork-in-the-road" (conditional) or "do-that-part-a-few-times" (loop) structure to our source code.

A variable is a labeled, local space that can contain a value. We refer to a variable by its label if we want to read or change that value. Values can be a few different types, such as *numbers* or *strings* or *booleans*. A string is a sequence of characters, and a boolean is simply a true/false value. JavaScript (also known as JS) automatically changes values from one type to another as needed.

A single-equals (=) is used to set values in variables, and can be combined with normal mathematical operators (**`+ - * /`**). The **==** operator compares two values, allowing JS to convert data types if needed; the **===** operator does *not* allow the types to be converted. We print to the developer console using the **`console.log`** function, which accepts a string (or converts inputs to a string).

We can examine the values of variables, and divert the flow of source code execution depending on those values, using the **`IF`** statement. This can be combined with an **`ELSE`**, to cover the other side of a conditional as well; these **`IF`**…**`ELSE`** statements can be nested. One can create compound comparisons using logical operators that represent *and*, *or* and *not* (**`&&`**, **`||`**, **`!`**).

To execute a specific piece of source code multiple times, there are two different types of *loops* available. A **`FOR`** loop is particularly useful when you know exactly how many times you need to loop; in other cases, a **`WHILE`** loop is simple and flexible. With both kinds of loops, you can use **`BREAK`** and **`CONTINUE`** statements to change the flow of code (to exit the loop or skip a certain iteration).

We can extract a piece of source code into a **`FUNCTION`** so that it can easily be called repeatedly. Functions (like variables) have labels, and to call a function, we list the function name with **`()`** following it. A function can require one or more values from outside, and we pass those values in by using parameters, which are included between the parentheses when calling the function, and similarly specified between the parentheses when defining the function.

The **`{`** and **`}`** characters are used to group code. How you choose to space (or compact) your source code will determine your personal ***coding style***.

Chapter 1 – Fundamentals

OK Ninjas-in-training, use your new knowledge. Can you solve these?

☐ Setting and Swapping

Set `myNumber` to `42`. Set `myName` to your name. Now swap `myNumber` into `myName` & vice versa.

☐ Print -52 to 1066

Print integers from -52 to 1066 using a `FOR` loop.

☐ Don't Worry, Be Happy

Create `beCheerful()`. Within it, `console.log` string `"good morning!"` Call it 98 times.

☐ Multiples of Three – but Not All

Using `FOR`, print multiples of 3 from -300 to 0. *Skip* -3 and -6.

☐ Printing Integers with While

Print integers from 2000 to 5280, using a `WHILE`.

☐ You Say It's Your Birthday

If 2 given numbers represent your birth month and day *in either order*, log `"How did you know?"`, else log `"Just another day...."`

☐ Leap Year

Write a function that determines whether a given `year` is a leap year. If a year is divisible by four, it is a leap year, unless it is divisible by 100. However, if it is divisible by 400, then it *is*.

☐ Print and Count

Print all integer multiples of 5, from 512 to 4096. Afterward, also log how many there were.

☐ Multiples of Six

Print multiples of 6 up to 60,000, using a `WHILE`.

☐ Counting, the Dojo Way

Print integers 1 to 100. If divisible by 5, print `"Coding"` *instead*. If by 10, also print `" Dojo"`.

☐ What Do You Know?

Your function will be given an input parameter `incoming`. Please `console.log` this value.

☐ Whoa, That Sucker's Huge…

Add odd integers from -300,000 to 300,000, and `console.log` the final sum. Is there a shortcut?

☐ Countdown by Fours

Log positive numbers starting at 2016, counting down by fours (exclude 0), *without* a `FOR` loop.

☐ Flexible Countdown

Based on earlier "Countdown by Fours", given `lowNum`, `highNum`, `mult`, print multiples of `mult` from `highNum` down to `lowNum`, using a `FOR`. For `(2,9,3)`, print `9 6 3` (on successive lines).

☐ The Final Countdown

This is based on "Flexible Countdown". The parameter names are not as helpful, but the problem is essentially identical; don't be thrown off! Given 4 parameters `(param1,param2,param3,param4)`, print the multiples of `param1`, starting at `param2` and extending to `param3`. One exception: if a multiple is equal to `param4`, then skip (don't print) it. Do this using a `WHILE`. Given `(3,5,17,9)`, print `6,12,15` (which are all of the multiples of 3 between 5 and 17, and excluding the value 9).

Chapter 1 – Fundamentals

Return Values

Parameters give functions a lot more flexibility. However, sometimes you don't want a function to do *all* the work; maybe you just want it to give you information so that then your code can do something based on the answer it gives you. This is when you would use the *return value* for a function.

Functions have names (usually). They (often) have parameters. They have code that will run when the function is executed. They generally have a *return value* as well, which is simply a value that is returned to the caller when the function finishes executing. Not all functions have *return* values, and looking at source code you might think that not all functions have a return statement. However, they indeed do, because if there is nothing stated, an implicit **`return`** is added automatically at the end of the function.

In JavaScript, if a caller "listens" to a function that ends with **`return`**, the caller receives **`undefined`**. If we want to be more helpful, we can explicitly return a value (for example, a variable or a literal). In other words, our function could **`return myNewName;`** or could **`return "Zaphod";`**. In either case, once the **`return`** statement runs, any subsequent lines of code in our function will *not* be executed. When program execution encounters a **`return`**, it exits the current function immediately.

If functions can return values, to *tell us the answer*, then whoever calls those functions must *listen for that answer*. It is easy enough to execute a function (**`greetSomeone(nameStr)`**, below) that has no return. If a function *does* return a value, then to actually *receive* that value the caller should save the result into a **`var`**, or otherwise "listen" to what it says (**`tellMeAGoodJoke()`**, below after):

```
// Calling a function that              // This one DOES return a value
// does NOT return a value              var aJoke = tellMeAGoodJoke();
greetSomeone("Claire");                 console.log(aJoke);
```

Above, **`tellMeAGoodJoke()`** returns a string which we copy into **`aJoke`** and display. See below for how to declare that function, but beware the function's sequel!

```
function tellMeAGoodJoke() {
   var jokeStr = "Have you heard about corduroy pillowcases?";
   jokeStr = jokeStr + " .... They're making headlines!";
   return jokeStr;
   jokeStr += "Thanks, I'm here all week...";   // this will never run!
}

// It may be a good joke, but it's a BAD FUNCTION. You only return once!
function tellMeAnotherOne() {
   var joke = "How many surrealists does it take to screw in a lightbulb?";
   return joke;
   return " .... A fish.";    // Wha? Oh I get it...but JavaScript won't.
}                             // Remember: you can't return twice!
```

Chapter 1 – Fundamentals

Arrays

An array is like a cabinet with multiple drawers, where each drawer stores a number, string, or even another array. In JavaScript, arrays are created by code like this:

```
var arr = [2, 4, 6, 8];   // create array with four distinct values
```

In our example, we created an array called **`arr`**. This array **`arr`** is like a file cabinet with three drawers. To look into one of these file cabinets, we have to specify which one. Each drawer is numbered, starting at the number 0 (not 1). The first drawer, drawer 0, has a value of 2; the second drawer, labeled 1, has a value of 4; the next drawer, which we call drawer 2, has a value of 6. In our code, we reference the different locations in an array by specifying the 'drawer number', which is really the offset from the beginning of our array. Specifically, we read an array value by putting its offset between square-brackets, as follows:

```
console.log(arr[1]);      // "4" (Not 2 - this is at arr[0])
```

Arrays have three important built-in properties: **`push`**, **`pop`** and **`length`**. We add a value to the end of our array (which lengthens it by one) with **`push`**:

```
arr.push(777);            // arr was [2,4,6,8], is now [2,4,6,8,777]
```

This pushes a new value *onto the end of the array*, so **`arr`** has a new value and is slightly longer – it is now **`[2,4,6,8,777]`**.

Similarly, we *remove (and return) the value at the end of the array* (and we shorten our array by one) by using the **`pop`** function:

```
var last = arr.pop();     // arr was [2,4,6,8,777], is now [2,4,6,8]
console.log(last);        // "777" - this is what pop() returned
```

The examples we've used above have lengthened and shorted our array. We see this on the page by just looking at all the values, but how would we quickly do this in code? We would use a useful property on every array called **`length`**. This is attached to each array like pop and push are, but it is not a function, so you do not need parentheses when using it:

```
console.log(arr);         // "[2,4,6,8]"
console.log(arr.length);  // "4" - vals are stored at indices 0,1,2,3
```

Said another way, **`arr.length`** is always one greater than **`arr`**'s highest populated index.

Writing Values into Arrays

In the previous example, our array **arr** had four (sometimes five) values. Each value in an array has its own space set aside for it, like the different drawers in a file cabinet.

```
var arr = [2, 4, 6, 8];  // create array with four distinct values
```

The beginning drawer (at index 0) has a value of 2; the second drawer (index 1) has a value of 4; the third drawer (index 2) has a value of 6. We change values within an array in the same way that we reference its values when reading from it: we enclose the index in square brackets, like this:

```
arr[1] = 10;             // arr was [2,4,6,8], is now [2,10,6,8].
```

This statement *sets* **arr[1]** *to be* **10**. It puts value **10** into **arr[1]**: index (drawer) **1** within **arr**.

We often need to swap the values of two variables (this will be handy later, for algorithms such as "reverse an array"). We can treat the spaces in an array exactly the same. What if we tried swapping the value at index 1 with the value at index 3? We might try something like the below:

```
// arr is currently [2,10,6,8]. We want to change it to [2,8,6,10]
x[1] = x[3];
x[3] = x[1];
console.log(x);          // ...but this code won't work quite right.
                         // arr got messed up! It is now [2,8,6,8].
```

The code above wouldn't quite work. For example, let's talk through this code step by step.

- Before starting, **arr** is equal to **[2,10,6,8]**.
- In line 2, we set **arr[1]** to be the value in **arr[3]**, which is **8**. Therefore, **arr** becomes **[2,8,6,8]**.
- When we run line 3, we set **arr[3]** to be the value in **arr[1]**, which is now **8**. Thus, we overwrite an **8** with an **8**, and **arr** remains **[2,8,6,8]**.

We can avoid this problem by creating a temporary variable to store the value of **arr[1]** before it is overwritten. To swap values (in an array or elsewhere), use a temporary variable. For example:

```
arr = [2, 10, 6, 8];
temp = arr[1];           // arr == [2,10,6,8], temp == 10
arr[1] = arr[3];         // arr == [2,8,6,8],  temp == 10
arr[3] = temp;           // arr == [2,8,6,10], temp == 10
console.log(arr);        // displays [2,8,6,10]
```

Success! Now, onward to algorithm challenges that use arrays.

Chapter 1 – Fundamentals

☐ Countdown

Create a function that accepts a number as an input. Return a new array that counts down by one, from the number (as array's 'zeroth' element) down to 0 (as the last element). How long is this array?

☐ Print and Return

Your function will receive an array with two numbers. Print the first value, and return the second.

☐ First Plus Length

Given an array, return the sum of the first value in the array, plus the array's length. What happens if the array's first value is *not* a number, but a string (like `"what?")` or a boolean (like `false`).

☐ Values Greater than Second

For `[1,3,5,7,9,13]`, print values that are greater than its 2nd value. Return how many values this is.

☐ Values Greater than Second, Generalized

Write a function that accepts any array, and returns a new array with the array values that are greater than its 2nd value. Print how many values this is. What will you do if the array is only one element long?

☐ This Length, That Value

Given two numbers, return array of length `num1` with each value `num2`. Print `"Jinx!"` if they are same.

☐ Fit the First Value

Your function should accept an array. If value at `[0]` is greater than array's length, print `"Too big!"`; if value at `[0]` is less than array's length, print `"Too small!"`; otherwise print `"Just right!"`.

☐ Fahrenheit to Celsius

Kelvin wants to convert between temperature scales. Create `fahrenheitToCelsius(fDegrees)` that accepts a number of degrees in Fahrenheit, and returns the equivalent temperature as expressed in Celsius degrees. For review, `Fahrenheit = (9/5 * Celsius) + 32`.

☐ Celsius to Fahrenheit

Create `celsiusToFahrenheit(cDegrees)` that accepts number of degrees Celsius, and returns the equivalent temperature expressed in Fahrenheit degrees.

(optional) Do Fahrenheit and Celsius values *equate* at a certain number? Scientific calculation can be complex, so for this challenge just try a series of Celsius integer values starting at 200, going downward (descending), checking whether it is equal to the corresponding Fahrenheit value.

Chapter 1 – Fundamentals

Combining Arrays and FOR Loops

In programming, it's very common to loop through each array value. We can do this as follows:

```
var nums = [1,3,5,7];                        // set up our loop
for (var idx = 0;idx < nums.length;idx++)  // for each index in arr...
{
   console.log(nums[idx]);                  // ...print the value
}
```

This prints each value in the array, using a FOR loop that iterates once for each array value.

What if we wanted an array with multiples of 3 up to 99,999? We accomplish this with the code below:

```
var arr = [];                              // create empty array
for (var val = 3;val <= 99999;val += 3)  // val will be 3,6,...99999
{
   arr.push(val);                          // add each val to arr
}
console.log(arr);                          // [3,6,9,12,..., 99999]
```

You will frequently write loops that, at each iteration's end, compare a variable (like **`idx`**) to the array's **`.length`**. Note: **`push()`** and **`pop()`** change an array's **`.length`**; if you need the original, save it off.

Here's an example of code that *<u>does not</u> work as the programmer intended:*

```
// BADCODE - intentionally buggy
function addEvenCount(arr) {
   // Count array even values & add that number to end of array
   for (var idx = 0; idx < arr.length; idx++) {
      if (idx == 0) {
         arr.push(0);              // First time, add 0 to end.
      }
      if (arr[idx] % 2 == 0) {     // Then just add to it as we go.
         arr[arr.length - 1] += 1;
      }
   }
}                                  // Oops! We counted the "2" as well.
```

The problem, of course, is that if we push our zero to the end of the array, that increments **`arr.length`**, and now our **`FOR`** loop will run on the index we just added as well. Given `[0,3,6,5]`, we want to change the array to `[0,3,6,5,2]`, but would instead change it to `[0,3,6,5,3]`.

Chapter 1 – Fundamentals

☐ Biggie Size

Given an array, write a function that changes all positive numbers in the array to “big”. Example: **`makeItBig([-1,3,5,-5])`** returns that same array, changed to **`[-1,"big","big",-5]`**.

☐ Print Low, Return High

Create a function that takes array of numbers. The function should print the lowest value in the array, and *return* the highest value in the array.

☐ Print One, Return Another

Build a function that takes array of numbers. The function should *print* second-to-last value in the array, and *return* **first odd** value in the array.

☐ Double Vision

Given array, create a function to return a *new* array where each value in the original has been doubled. Calling **`double([1,2,3])`** should return **`[2,4,6]`** without changing original.

☐ Count Positives

Given array of numbers, create function to replace last value with number of positive values. Example, **`countPositives([-1,1,1,1])`** changes array to **`[-1,1,1,3]`** and returns it.

☐ Evens and Odds

Create a function that accepts an array. Every time that array has three odd values in a row, print **`"That’s odd!"`** Every time the array has three evens in a row, print **`"Even more so!"`**

☐ Increment the Seconds

Given **`arr`**, add 1 to odd elements (**`[1]`**, **`[3]`**, etc.), **`console.log`** all values and return **`arr`**.

☐ Previous Lengths

You are passed an array containing strings. Working within that same array, replace each string with a number – the length of the string at *previous* array index – and return the array.

☐ Add Seven to Most

Build function that accepts array. Return a new array with all values *except first*, adding 7 to each. Do not alter the original array.

☐ Reverse Array

Given array, write a function to reverse values, in-place. Example: **`reverse([3,1,6,4,2])`** returns same array, containing **`[2,4,6,1,3]`**.

☐ Outlook: Negative

Given an array, create and return a new one containing all the values of the provided array, made negative (*not simply multiplied by -1*). Given **`[1,-3,5]`**, return **`[-1,-3,-5]`**.

☐ Always Hungry

Create a function that accepts an array, and prints **`"yummy"`** each time one of the values is equal to **`"food"`**. If no array elements are **`"food"`**, then print **`"I'm hungry"`** once.

☐ Swap Toward the Center

Given array, swap first and last, third and third-to-last, etc. Input **`[true,42,"Ada",2,"pizza"]`** becomes **`["pizza",42,"Ada",2,true]`**. Change **`[1,2,3,4,5,6]`** to **`[6,2,4,3,5,1]`**.

☐ Scale the Array

Given array **`arr`** and number **`num`**, multiply each **`arr`** value by **`num`**, and return the changed **`arr`**.