Algorithms Course Summary

Onsite Overview

Every day from **9:00AM to 10:00PM**, we will have an algorithm challenge. Each cohort will be divided into groups, and those groups will be tasked with solving the algorithm on a whiteboard. In the last 15 minutes of the session, we'll have one group present solutions, as will your instructor/TA.

Goals

One important goal of this course is to get you comfortable clearly describing the functionality of your code. This is important for technical interviews, where you will be asked to demonstrate your knowledge using only a whiteboard.

Another important goal is to familiarize yourself with the 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 start thinking like a computer is through *repetition*. Make sure you're here at 9:00AM every session, to get the challenge's introduction.

2. No Laptops

To simulate a technical interview, do not use your laptop or refer to old code during these challenges. Always work out problems on a whiteboard, unless directed by the instructor.

3. Be respectful

Being able to walk through your algorithm and explain how it works is as important as a correct solution. There will be many chances for you to discuss with peers or present to the class. All students should give full attention and respect to anyone speaking. Public speaking is a very common fear, and we must learn to conquer our nerves. This can only happen in a welcoming environment.

4. Work in groups

Working in groups provides excellent practice articulating your thoughts. It's important to be comfortable describing your code, a skill that's not only useful for technical interviews, but also when working in teams of developers more generally. Unless otherwise directed by the instructor, solve the algorithm challenges in groups ***no smaller than two and no larger than three***.

Presenting

Every day, two groups will present their solutions. Make sure everybody in your group has a chance to describe the algorithm, and give presenting groups the proper respect.

Questions to ask about solutions

1. Is it clear and understandable?

Can you easily explain its 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 variable 'x'?

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. If an algorithm contains duplicate code, pull that code into helper functions. Finally, to repeat: remember the acronym DRY. (-:

4. Is it efficient?

Does your function contain only the necessary statements, and does it require only the necessary memory for additional variables? Does it stay efficient (in run time as well as memory usage), as the size of inputs 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 into your thinking. You may even get help if you are on the wrong track!
- Describe your assumptions. Clarify before writing code.
- List sample inputs, along with expected outputs. This is a great way to validate your understanding of the problem.
- Don't get stuck. Add comments/pseudocode and move on.
- Break big problems down, into smaller problems.
- Focus less on 'correct' solutions; focus on correct outputs.
- Don't stress! Morning algorithms is about brain cardio. This is not a constant evaluation of your abilities or expertise!
- Have fun! Consider these as simply puzzles that make you a better, more well-rounded developer.

front
length
average
min, max
display
back
removeBack
addBack
removeVal
prependVal
appendVal
splitOnVal
partition
deleteGivenNode
dedupeList
dedupeListWithoutBuffer

The "Bug-Laden 13" (#3)

Chapter 5 – Queues and Stacks

Queues

Enqueue
Front
Is Empty
Dequeue
Contains
Size

Stacks

Stack Implementation Based on Array
Stack Push
Stack Top
Stack Is Empty
Stack Pop
Stack Contains
Stack Size
Stack / Queue Code-Sharing
Deque Class Implementation

Circular Queues

Enqueue
Front
Is Empty
Is Full
Dequeue
Contains
Size

Contains
Max
Is Empty

Self-Instantiating Classes
Remove
Is Valid
RemoveAll
Add Without Dupes

Binary Tree Depth
Height
Is Balanced

Binary Search Tree Traversal
Bst2Array
Lowest Common Ancestor
Bst2List
Traverse Pre-Order Without Recursion
Val Before
Node Before
Val After
Node After

Making BST a Fully Navigable Data Structure
BST With Parent
(TBD) Where's the Bug? (tree version)

Chapter 11 – Sorts
Bubble Sort Array
Bubble Sort List
(TBD) Multikey Sort
Selection Sort Array
Selection Sort List

Big-O Notation
Insertion Sort
Partition Array
Insertion Sort List
Partition List

Adaptivity
QuickSort (array)
Combine Sorted Arrays
Partition3

Stability
Combine Sorted Lists
Merge Sort List

Trie MultiSet

Trie MultiSet Insert
Trie MultiSet Remove
Trie MultiSet Contains
Trie MultiSet Size
Trie MultiSet Auto Complete

TrieMap

TrieMap Insert
TrieMap Contains
TrieMap Remove
TrieMap Size
TrieMap First
TrieMap Last
TrieMap Next

(TBD) Graphs

(TBD) Adjacency List
(TBD) Adjacency Map

(TBD) Depth-First Search

(TBD) Path Exists
(TBD) Someone on the Inside

(TBD) Breadth-First Search

(TBD) Shortest Path
(TBD) Graphs: Seven Degrees of Kevin Bacon

(TBD) Directed Acyclic Graph

(TBD) Is DAG
(TBD) DAG2Arr
(TBD) Word Ladder
(TBD) Where's the Bug? (trie/graph version)

(TBD) Chapter 16 – Optimization and Estimation

(TBD) Deck Of Cards
(TBD) Speed Prime
(TBD) Optimizing N-Queens

Estimation

Piano Tuners
Gas Stations
Kindergarten Teachers
Earth's Circumference
Weight of a Ferry
Basketballs in a 747

Black Belt Exam

Chapter 17 – Bit Arithmetic

Numerical Systems

Octal System
- Decimal to Octal Practice
- Octal to Decimal Practice
- Decimal to Octal String
- Octal String to Value

Hexadecimal System
- Decimal to Hexadecimal
- Hexadecimal to Decimal
- Decimal to Hexadecimal String
- Hexadecimal String to Value

Binary System
- Decimal to Binary
- Binary to Decimal
- Decimal to Binary String
- Binary String to Value

Bitwise Operators, Part 1
- Bitwise AND
- Bitwise OR
- Bitwise NOT

Bitwise Operators, Part 2
- Bitwise XOR
- Bitwise LSL
- Bitwise LSR

Bit Shifting and Masking
- CountSetBits
- EncodeBytesTo32
- Decode32ToBytes
- ByteArray
- EncodeBitNum
- DecodeBitNum
- BitArray
- RadixSort2
- Sprinklers
- LED Encoding

Chapter 18 – Trees, Part III

AVL Trees
- Height (AVL)

Chapter 0 – Foundation Concepts

What is Source Code?

Computers are amazing. They can rapidly perform complex calculations, store immense amounts of data, and almost instantly retrieve very specific bits of information out of those enormous amounts of data. In addition to being really fast, sometimes they seem almost human (or even superhuman) in their ability to use information to make decisions. How do they do it?

Computers only do what they are told by programmers. How do we, as software engineers, teach computers what we want them to do? We create a sequence of instructions, provide it to the computer, and the computer "mindlessly" runs those instructions. Computers do not naturally understand human language, and humans do not understand the language that computers use, so we use some sort of go-between language that humans can understand, that is *then* translated into machine language. Many of these "go-between" languages have been created -- PHP, Python, Ruby, JavaScript, Swift, C, Java, Erlang, Go, and many dozens of others. Each has different strengths and is useful in different situations, and more are created all the time. Languages all essentially do the same thing though: they take a series of steps that a human has laid out to instruct the computer how to respond, and they translate those steps into a format that the computer can understand and can later execute.

These series of steps, written in human-readable languages like PHP, are what we call ***Source Code***. The translation from source code into *machine code* is done differently across programming languages. *Interpreted languages* like PHP, Python and Ruby will translate from source code into machine code "on the fly" immediately before the computer needs it. Ultimately the computer simply follows the instructions that it was given -- it executes (runs) machine code that was translated (*built*) from some piece of source code.

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 these later.

Chapter 0 – Foundation Concepts

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 we have stored there, we simply need to refer to our label *myName*.

In most programming languages, you will see **=** as well as **==**. These mean two different things! The single-equals can be described as "Set the value of X equal to the value of Y". The double-equals can be described as "Is the value of X equivalent to the value of Y?" In short, **=** *sets things*, and **==** *compares things*.

Linking to the next 'Printing' topic, if you want to display the value of a variable *num*, you would call `console.log(num)`.

Chapter 0 – Foundation Concepts

Printing

Eventually, you will create fabulous web systems and/or applications that do very fancy things with graphical user interface. However, when you are first learning how to program, you start with having your programs write simple text messages 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 send a text string to the developer console, which is where errors, warnings and other messages about our program go as well.

To log a message to the console, we will use the **`console.log();`** command. Between the parentheses of this call, we can put any message we want to be displayed. This could be a literal or a variable – something like this:

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

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

Chapter 0 – Foundation Concepts

Conditionals

If you are driving and get to a fork in the road, you have to make a decision about which way to go. Most likely, you will decide which way to go based on some very good reason. In your code, there is a mechanism like this too. You can use an **`IF`** statement to look at the value of a variable or perhaps compare two variables, and execute certain lines of code based on that result. If you wish, you can also execute *other* lines of code if the result goes the other way. The important point here is that each decision has only two possible outcomes -- you have a certain test or comparison that you do, and IF the test passes then you execute certain code. If you wish, you can also have a different set of code that you execute in the "test did not pass" case (called the **`ELSE`**). 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 are comparing the value of variable **`myName`** to the string **`"Martin"`**. If that comparison passes, then we will execute the next section of code, within the curly braces. Otherwise, we will skip those lines of code. What if we want to greet the user with some other cheerful comment, even if the user was not Martin? If you wish, you can also have a different set of code that you execute in the "test did not pass" 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!");
}
```

Two other notes. First, the *"test"* that is between the parentheses for these IF statements can be more than just a single comparison. You can create compound tests, using a combination of the logical AND, OR and NOT connectors just like you might in natural language. In JavaScript, OR is ||, so a statement like "*if it is raining or if it is too far to walk, then let's call Uber!*" could become this source code:

```
if (raining == true || distanceMiles > 2)  {  callUber( );  }
```

Just as in our natural language statement, our code would execute the **`callUber()`** function, unless both tests are not true (i.e. if it is not raining *and* distance is two miles or less.

Second, it is possible for us to nest **`IF..ELSE`** statements inside other **`IF..ELSE`** statements, or chain them together (such as **`IF..ELSE IF..ELSE`**).

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*.

Chapter 0 – Foundation Concepts

FOR Loops

FOR loops are good when you know exactly how many times those lines of code should be run. *WHILE loops* are a combination of loops with conditionals; they are good when you don't know how many times to loop, but you know you want to loop *until* a certain test is true (or more accurately, if you want to continue looping *while* a certain test continues to be true). To create a *FOR loop*, in addition to the chunk of code to be looped, you need to specify three things, and these things go into the parentheses after the FOR: any initial setup, a test that needs to be true in order to continue looping, and any code that should be run at the end of each time through the loop. Here is an annotated example:

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

How does this *FOR loop* really work? Up front, the local variable *'num'* is created and is set to a value of 1. This step A happens exactly once, and then our loop starts. Step B: *'num'* is compared to 6. If it is less than 6, then (step C) the chunk of code within the curly braces is executed, and after that (step D) 1 is added to *num*. Then we go back to step B. When the test at B fails for the first time, we immediately exit without executing the code in C or D. At that point, code execution continues onward from E, the point immediately following the close-curly-brace.

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.

Chapter 0 – Foundation Concepts

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:

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

This WHILE code will execute in this sequence: A - B-C-D - B-C-D - B-C-D - B-C-D - B-C-D - B - E.

Chapter 0 – Foundation Concepts

Other Loop Tips

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

```
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.`

```
var counter=5;
counter=counter-1;
counter--;
// counter now holds a value of 3
```

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

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

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 a FOR loop is critical, so get really familiar with this.

Chapter 0 – Foundation Concepts

Loops and Code Flow

With more complex loops, you might discover that you need to break out of a loop early, or instead to skip the rest of the current pass but continue looping. In JavaScript, there are the special *BREAK* and *CONTINUE* keywords that allow you to do this. If you add **`break;`** to your code, program execution will immediately exit the specific loop that you are currently in, and continue execution immediately following the loop. Even the final end-loop statement (**`num = num + 1`** above) will not be executed. If you add **`continue;`** to your code, the rest of that pass through the loop will be skipped but the loop-end statement *is* executed and looping will continue. See examples below.

The following code will print the first two lines, but then will immediately exit.

```
var num = 1;
while (num < 5)  {
      if (num == 3) {
            break;
      }
      console.log("I'm counting! The number is ", num);
      num = num + 1;
}
```

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

The following code will count from 1 to 4, printing a statement about each number, but will completely forget to say anything about 3, because when num == 3, the **`continue`** forces it to skip the rest of that loop and continue from the top of the loop (after incrementing *num*).

```
for (var num = 1; num < 5; num += 1)  {
      if (num == 3) {
            continue;
      }
      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

`Continue` is uncommon, but you'll see **`break`** everywhere. Get comfortable setting up loops that (if things go well) execute for a certain number of iterations, but at any time could exit if you encounter a certain condition. With **`break`**, it isn't preposterous to see an infinite **`while (true)`**! My goodness.

Chapter 0 – Foundation Concepts

Arrays

An array is like a cabinet with multiple drawers, where each drawer can store information: a number, string, or even another array. In JavaScript, an array is created by doing something like below:

```
var x = [1,3,5];
```

In this example, we created an array called x. x is like a cabinet with three drawers, the first drawer (also called drawer with index 0) has a value of 1; the second drawer (index 1) has a value of 3; the third drawer (index 2) has a value of 5. We can set a value in x to be something else by doing:

```
x[1] = 10;
```

This means 'SET x[1] to be 10' or in other words, put a value of 10 into x[1] (into the x cabinet, at the index 1). This updates x to be [1,10,5]. We can add a new member in the array by doing:

```
x.push(777);
```

This pushes a new value onto the end of the array. Array x becomes [1, 10, 5, 777]. We often need to swap the values of two variables (this will be handy later, for algorithms such as "reverse an array"). What if we tried swapping the second value with the third value by doing something like below:

```
x = [1,3,5,7];
x[1] = x[3];
x[3] = x[1];
console.log(x);   // this code won't work quite right
```

The code above wouldn't quite work. For example, let's run the code in our mind.

- When we run line 1, x is set to be [1,3,5,7].
- In line 2, we set x[1] to be the value in x[3], which is 7. In other words, x becomes [1,7,5,7].
- When we run line 3, we set x[3] to be the value in x[1]: 7. In other words, x becomes [1,7,5,7].

We avoid this by creating a temporary variable to store the value before it is overwritten. For example:

```
x = [1,3,5,7];
temp = x[1];
x[1] = x[3];
x[3] = temp;
console.log(x);
```

To swap values in the array (or elsewhere), always use a temporary variable.

Chapter 0 – Foundation Concepts

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 (idx = 0;idx < nums.length;idx++) // for each index in array...
{
      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 generate this with the code below:

```
var arr = [];                          // create empty array
for (val = 3;val <= 99999;val = val+3)// generate values 3,6,...99999
{
      arr.push(val);                   // add each values to our array
}
console.log(arr);                      // [3,6,9,12,15,18, ..., 99999]
```

Chapter 0 – Foundation Concepts

Functions

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

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

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

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

```
printMyName();
```

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.

Again, when we *declare* a function, that just makes it so that some other *caller* can execute our function from some other place in the code, at some other time. It does not run the function immediately.

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 printMyName function does, in fact, call the built-in console.log function. In fact, any code you might write only starts running when some other code (maybe part of the computer browser or the operating system) calls it; it is the same as if your function might call some other. So, except for the very first piece of code that runs when the computer starts up, all other code is running only because some other code called it.

Chapter 0 – Foundation Concepts

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 clearer and safer to include these, just in case you add more code to your loop later.

Chapter 0 – Foundation Concepts

Return Values

Having parameters gives a function a lot more flexibility. However, sometimes when you call a function you don't want it 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 say that not all functions have a return statement. However, they *do* indeed, because if there is none, 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 we create functions that return values in order to "tell us the answer", then whoever calls those functions needs to "listen for that answer" as well. Let's say that we execute a function, like this:

```
greetSomeone("Claire");
```

Function *greetSomeone* returns no value, which is good because we are not listening! (poor Claire) However, if we call a function that returns a value, then to actually receive that return value we execute it like this:

```
var joke = tellMeAGoodJoke();
console.log(joke);
```

In this case, the *tellMeAGoodJoke( )* would presumably return a string, and whatever its value, we have copied it into a local variable labelled *joke* and then printed it.

Chapter 0 – Foundation Concepts

Comments

Source code containing good comments is a joy to work with. You don't spend as much time trying to figure it out, because the creator cared enough to spend just a few moments ahead of time to explain it. There are two ways to write comments in JavaScript source code.

One option: `//`. After a double-slash, the *rest of that line* is a comment. Your code might look like this:

```
function greetSomeone(person) {
      if (person == "Martin")     //  Check whether it is Martin...
      {
            console.log("Yo dawg, howz it goin?");
      }
      else                        // ...if not, probably a normal human
      {
            console.log("Greetings Earthling!");
      }
}
```

Second option: `/* ... */` . These /* and */ bookends can span multiple lines, and *everything between them* is considered a non-source-code comment. This style would look like this:

```
/*    Simple function that responds directly if
      person is Martin, otherwise it provides a
      more generic salutation. No return value.
*/
function greetSomeone(person)
{
      if (person == "Martin") {
            console.log("Yo dawg, howz it goin?");
      } else {
            console.log("Greetings Earthling!");
      }
}
```

Both are commonly used. Quick comments sprinkled throughout your course are usually `//`; larger comment blocks are often `/* … */`. The main thing is simply to add a few comments. The next person to work with the code (perhaps YOU in the future when you have forgotten details) will appreciate it.

Now that you have been introduced to the foundation concepts of source code execution, variables, conditionals, loops, arrays, functions, parameters, return values and comments, you are ready to continue onward to the rest of the algorithm materials. Enjoy!

The “Basic 13”

The foundation “Basic 13” algorithm challenges.

Print 1-255

Print all the integers from 1 to 255.

Print Sum 0-255

Print integers from 0 to 255, and with each integer print the sum so far.

Find and Print Max

Given an array, find and print its largest element.

Array with Odds

Create an array with all the odd integers between 1 and 255 (inclusive).

Greater Than Y

Given an array and a value Y, count and print the number of array values greater than Y.

Max, Min, Average

Given an array, print the max, min and average values for that array.

Swap String For Array Negative Values

Replace any negative array values with `'Dojo'`.

Print Odds 1-255

Print all odd integers from 1 to 255.

Iterate and Print Array

Iterate through a given array, printing each value.

Get and Print Average

Analyze an array's values and print the average.

Square the Values

Square each value in a given array, returning that same array with changed values.

Zero Out Negative Numbers

Return the given array, after setting any negative values to zero.

Shift Array Values

Given an array, move all values forward by one index, dropping the first and leaving a `'0'` value at the end.

Chapter 1 – Fundamentals

This chapter, you will familiarize yourself with basic constructs of programming: loops, conditionals, logic operators, and a few techniques.

It is imperative that you complete the mandatory challenges from earlier in the bootcamp.

Here is a list of concepts to study. Some or all will be used to solve this chapter's challenges.

variables, functions for loops, while loops
conditional (if / else) statements
console.log return values
math.random, math.floor, math.ceil
&& || ! (and, or, not) % (modulus)

Being able to write a t-diagram to keep track of your variables while you write out an algorithm by hand is extremely beneficial. You should use a t-diagram for algorithm challenges this chapter.

To put it simply. test-driven development (TDD) is a design technique where you must **first write a test that fails** before you write any new code, with the **goal of writing clean code that passes the test**. We will supply tests, if/else checks, and sample input/outputs to help guide your solutions.

At the bottom of the page are some examples of simple Javascript constructs we'll use in this chapter. Remember these basic building blocks!

For Loops

```
for (INITIALIZATION; TEST; INCREMENT/DECREMENT)  {
    // BODY of the loop - runs repeatedly while TEST is true
    // INIT. TEST?-BODY-INCREMENT. TEST?-BODY-INCREMENT. TEST?-[exit]
}
```

Conditional (if / else) statements

```
if (CONDITION_1 && CONDITION_2)  {
// body of 'IF' – runs if CONDITION_1 and CONDITION_2 are BOTH true
}
else   {
// body of 'ELSE' – runs if CONDITION_1 -OR- CONDITION_2 are false
}
```

Functions

```
// Declaring standalone functions
function MY_FUNCTION(PARAMETER_1, PARAMETER_2)
{
    // body of function - only runs when function is invoked
}

// Calling standalone functions.
// ARGUMENTS passed by callers enter functions as PARAMETERS
```

```
MY_FUNCTION(ARGUMENT_1, ARGUMENT_2);
```

Chapter 1 – Fundamentals

What is a variable? Think of this as simply an empty container with a label. Once you put a value into the container, you can refer to this value by the label. Put a value into a variable by using single-equals, which you can read as “is set to a value of”. In other words, code **`var name = "Zaphod"`** can be read as “Variable labeled ‘name’ is set to a value of Zaphod”. After this line of code, when you refer to ‘name’, you get a value of “Zaphod”. If you are still getting used to the idea of variables, *don’t panic*.

Sigma

Implement a function **`sigma(num)`** that, given a number, returns the sum of all positive integers from 1 up to number (inclusive). Ex.: **`sigma(3)`** = 6 (or 1+2+3); **`sigma(5)`** = 15 (or 1+2+3+4+5).

Factorial

Write a function **`factorial(num)`** that, given a number, returns the product (multiplication) of all positive integers from 1 up to number (inclusive). For example, **`factorial(3)`** = 6 (or 1 * 2 * 3); **`factorial(5)`** = 120 (or 1 * 2 * 3 * 4 * 5).

Chapter 1 – Fundamentals

This chapter you will familiarize yourself with basic programming constructs. Some or all of the below concepts will be used in this section's challenges.

for loops, while loops *if / else statements* *% (called modulus)*
Math.random, Math.floor, Math.ceil *console.log*

Threes and Fives

Create function **`ThreesFives()`** that adds each value from 100 and 4000000 (inclusive) if that value is evenly divisible by 3 or 5 *but not both*. Display the final sum in the console.

Second: Change your function to make a **`BetterThreesFives(start,end)`** where *start* and *end* values are customizable. You can think of the above **`ThreesFives()`** function as **`BetterThreesFives(100,4000000)`**.

Generate Coin Change

Implement **`generateCoinChange(cents)`** that accepts a parameter for the number of cents, and computes how to represent that amount with the smallest number of coins. Console.log the result.

Chapter 1 – Fundamentals

This chapter you will familiarize yourself with basic programming constructs. Here is a list of methods for you to study. Some or all of these will be used to solve this chapter's challenges.

for loops, while loops *if / else statements* *% (called modulus)*
Math.random, Math.floor, Math.ceil *console.log*

Statistics to Doubles

Implement a 'die' that randomly returns an integer between 1 and 6 inclusive. Roll a pair of these dice, tracking the statistics until doubles are rolled. Display the *number of rolls*, *min*, *max*, and *average*.

Sum To One Digit

Implement a function **`sumToOne(num)`** that, given a number, sums that number's digits repeatedly until the sum is only one digit. Return that final one digit result.

Chapter 1 – Fundamentals

This chapter you will familiarize yourself with basic programming constructs. Here is a list of methods for you to study. Some or all of these will be used to solve this chapter's challenges.

for loops, while loops *if / else statements* *% (called modulus)*
Math.random, Math.floor, Math.ceil *console.log*

Fibonacci

Implement the Fibonacci function, a famous mathematical equation that generates a numerical sequence such that each number is the sum of the previous two. The Fibonacci numbers at index 0 and 1, coincidentally, have values of 0 and 1. Your function should accept an argument of which Fibonacci number.
Examples: **`fibonacci(2)`** = 1, **`fibonacci(3)`** = 2, **`fibonacci(4)`** = 3, **`fibonacci(5)`** = 5, etc.

Chapter 1 – Fundamentals

Data Sufficiency

One important concept for solving algorithms is the idea of *data sufficiency*. In an algorithm challenge as well as in real-world problems, just because you are given a piece of data doesn't necessarily mean that it is important, and sometimes your code will run much faster if you can discard that additional data. In fact, sometimes your code cannot run at all *unless* you let go of unneeded data!

This chapter explored basic programming constructs. Here are additional problems to keep you busy!

Last Digit of A to the B

Modern computers can handle very large numbers, but this ability has limits. If a number is repeatedly multiplied by itself, it eventually exceeds the computer's ability to accurately represent it. (Side note: the number of times it is multiplied by itself is called the *exponent*.) For an optional end-of-chapter challenge, determine the smallest (least significant) digit of a number that is potentially very, very large. You may find that you must do this without computing the actual (unimaginably large) number.

Implement a function that accepts two non-negative integers as arguments. Function **lastDigitAtoB(a, b)** should return the last digit of the first number (a) raised to an exponent of the second number (b).

Examples: given `(3, 4)`, you should return `1` (the last digit of 81: 3 * 3 * 3 * 3). Given `(12, 5)`, return `2` (the least significant digit of 248832, which is 12 * 12 * 12 * 12 * 12).

Clock Hand Angles

Traditional clocks are increasingly uncommon, but most can still read rotating hands of hours, minutes, and seconds.

Create function **clockHandAngles(seconds)** that, given the number of seconds since 12:00:00, will print the angles (in degrees) of the hour, minute and second hands. As a quick review, there are 360 degrees in a full arc rotation. Treat “straight-up” 12:00:00 as 0 degrees for all hands.

“Basic 13” Review

Solutions for the “Basic 13” algorithm challenges.

Print 1-255

Print all the integers from 1 to 255.

```
function print1to255()
{
  var num = 1;
  while (num <= 255) {
    console.log(num);
    num = num + 1;
  }
}
```

Print Sum 0-255

Print integers from 0 to 255, and the sum so far.

```
function printSum1to255()
{
  var sum = 0;
  for (var num = 0; num <= 255; num++) {
    sum += num;
    console.log("New number:", num, "Sum:", sum);
  }
}
```

Find and Print Max

Print the largest element in a given array.

```
function printArrayMax(arr)
{
  if (arr.length == 0) {
    console.log("Empty array, no max value.");
    return;
  }

  var max = arr[0];
  for (var idx = 1; idx < arr.length; idx++) {
    if (arr[idx] > max) {
      max = arr[idx];
    }
```

```
  }
  console.log("Max value is:", max);
}
```

Print Odds 1-255

Print all odd integers from 1 to 255.

```
function printOdds1to255() {
  var num = 1;
  while (num <= 255) {
    console.log(num);
    num = num + 2;
  }
}
```

Iterate and Print Array

Print all values in a given array.

```
function printArrayValues(arr) {
  for (var index = 0;index < arr.length;index++) {
    console.log("array[", index, "] is equal to", arr[index]);
  }
}
```

Get and Print Average

Analyze an array's values and print the average.

```
function printArrayAverage(arr) {
  if (arr.length == 0) {
    console.log("Empty arr, no average val");
    return;
  }

  var sum = arr[0];
  for (var idx = 1; idx < arr.length; idx++) {
    sum += arr[idx];
  }
  console.log("Average value is:", sum / arr.length);
}
```

Array with Odds

Create an array with odd integers from 1-255.

```
function oddArray1to255() {
  var oddArray = [];
  for (var num = 1; num <= 255; num += 2) {
    oddArray.push(num);
  }
  return oddArray;
}
```

Greater Than Y

Count and print the number of array values less than a given Y.

```
function numGreaterThanY(arr, y) {
  var numGreater = 0;
  for (var idx = 0; idx < arr.length; idx++) {
    if (arr[idx] > y) {
      numGreater++;
    }
  }
  console.log("%d values are greater than %d", numGreater, y);
}
```

Max, Min, Average

Given an array, print max, min and average values.

```
function maxMinAverage(arr) {
  if (arr.length == 0) {
    console.log("Null arr, no min/max/avg");
    return;
  }
  var min = arr[0];
  var max = arr[0];
  var sum = arr[0];
  for (var idx = 1; idx < arr.length; idx++) {
    if (arr[idx] < min) {
      min = arr[idx];
    }
    if (arr[idx] > max) {
      max = arr[idx];
    }
    sum += arr[idx];
  }
  console.log("Max:", max, " Min:", min);
  console.log("Avg value:", sum / arr.length);
}
```

Square the Values

Given an array, square each value in the array.

```
function squareArrVals(arr) {
  for (var idx = 0; idx < arr.length; idx++) {
    arr[idx] = arr[idx] * arr[idx];
  }
  return arr;
}
```

Zero Out Negative Numbers

Set negative array values to zero.

```
function setNegsToZero(arr)
{
  for (var idx = 0; idx < arr.length; idx++)
  {
    if (arr[idx] < 0)
    {
      arr[idx] = 0;
    }
  }
  return arr;
}
```

Shift Array Values

Shift array values: drop the first and leave `'0'` at end.

```
function arrShift(arr)
{
  for (var idx = 1; idx < arr.length; idx++)
  {
    arr[idx - 1] = arr[idx];
  }
  arr[arr.length - 1] = 0;
  return arr;
}
```

Swap String For Array Negative Values

Replace any negative array values with `'Dojo'`.

```
function numToStr(arr)
{
  for (var idx = 0; idx < arr.length; idx++)
  {
    if (arr[idx] < 0)
    {
      arr[idx] = "Dojo";
    }
  }
  return arr;
}
```

Chapter 2 – Arrays

This chapter, we explore the *array* data structure: reading, changing, adding and removing elements (hence changing the array's length). Before chapter's end, we will touch upon associative arrays as well. At this point we expect you to quickly complete the 13 mandatory algorithm challenges. Also, building Array functions such as min(), max(), sum() and average() should be easy and rapid.

Arrays store multiple values, which are accessed by specifying the *index* (the offset from the front of the array) in square brackets. This *random-access* characteristic makes arrays well-suited for accessing values in a different order than they were added. Arrays are less suitable (but still commonly used) in scenarios with many insertions and removals. Each value must be moved to create space for an insertion (or to fill a vacancy caused by a removal). JS array values can be different data types: one array can contain numbers, booleans, strings, etc.

Arrays are zero-based: an array's first value is located at index 0. Accordingly, the array attribute *length* means "one more than the last index." As with other interpreted languages, JavaScript arrays are not fixed-length; they automatically grow as values are set beyond the current length.

Tracking variables with T-diagrams is extremely beneficial. Use a t-diagram for challenges this chapter. Below are some of the constructs we'll use this chapter. Remember these basic building blocks!

Declaring a new array:

```
var myArr = [];
console.log(myArr.length);      // -> "0"
```

Setting and accessing array values:

```
myArr[0] = 42;                  // myArr = [42], myArr.length == 1
console.log(myArr[0]);          // -> "42"
```

Array.length is determined by the largest index:

```
myArr[1] = "hello";            // myArr == [42,"hello"], myArr.length == 2
myArr[2] = true;               // myArr == [42,"hello",true], myArr.length == 3
```

Overwriting array values:

```
myArr[0] = 101;                // myArr == [101, "hello", true]
```

Arrays can be sparsely populated:

```
myArr[myArr.length+2]=0.2;     // myArr == [101,"hello",true,undefined,0.2]
console.log(myArr.length);     // -> "5"
```

Shorten an array with the pop() method:

```
myArr.pop();                   // myArr == [101,"hello",true,undefined]
```

```
myArr.pop();                    // myArr == [101,"hello",true], myArr.length==3
```

Chapter 2 – Arrays

This chapter you will familiarize yourself with basic array manipulation. From your work with the Basic 13 challenges, we assume that you already know how to read from numerical arrays, and that you can easily create JavaScript functions to get the **minimum** or **maximum** value, the **sum** of all values in the array, or the **average** of all values in the array. If this is not the case, then definitely review those implementations before continuing to today's challenges.

Here is a list of concepts to consider; some or all will be used in this chapter.

for / while loops *array.pop() & push()*
arrays grow: arr.length == lastIdx-1 *if / else statements*
can contain different data types *arrs are objects, passed by reference (ptr)*

PushFront

Given an array and an additional value, *insert this value* at the beginning of the array. Do this without using any built-in array methods.

PopFront

Given an array, *remove and return the value* at the beginning of the array. Do this without using any built-in array methods except `pop()`.

InsertAt

Given an array, index, and additional value, *insert the value into the array* at the given index. Do this without using built-in array methods. You can think of `PushFront(arr,val)` as equivalent to `InsertAt(arr,0,val)`.

RemoveAt

Given an array and an index into the array, *remove and return the array value* at that index. Do this without using any built-in array methods except `pop()`. Think of `PopFront(arr)` as equivalent to `RemoveAt(arr,0)`.

Chapter 2 – Arrays

Passing By Reference

Arrays are passed by *reference*. This means that a pointer to that array is what is sent to another function, when an array is sent as an argument. Therefore, even though a parameter is always a copy of the original, with arrays (and all object types) what is copied is the pointer, which results in the caller and the callee both having a copy of the same pointer, hence both are looking at the same location in memory, and hence both will reference the same array. When you pass an array to another function, that array is "live" – changes the callee makes in that array will be reflected when we return to the caller, regardless of whether the called function returns that array.

This chapter you will familiarize yourself with basic array manipulation. Here is a list of concepts to study; some or all will be used in this chapter's challenges.

for / while loops *array.pop() & push()*
arrays grow: arr.length == lastIdx-1 *if / else statements*
can contain different data types *arrs are objects, passed by reference (ptr)*

Reverse Array

Given a numerical array, reverse the order of the values. The reversed array should have the same length, with existing elements moved to other indices so that the order of elements is reversed.

Remove Negatives

Implement a function **`removeNegatives()`** that accepts an array and removes any values that are less than zero.

Second-level challenge: don't use nested loops.

Skyline Heights

You are given an array with heights of consecutive buildings in the city. For example, `[-1,7,3]` would represent three buildings: first is actually below street level, second is seven stories high, and third is three stories high (but hidden behind the seven-story onbe). You are situated at street level. Return an array containing heights of the buildings you can see, in order. Given `[1,-1,7,3]` return `[1,7]`.

Chapter 2 – Arrays

This chapter you will familiarize yourself with basic array manipulation. Here is a list of methods to study; some or all will be used in this chapter's challenges.

for / while loops *array.pop() & push()*
arrays grow: arr.length == lastIdx-1 *if / else statements*
can contain different data types *arrs are objects, passed by reference (ptr)*

Binary Search

Given a sorted array and a value, return whether that value is present in the array. Do not sequentially iterate the entire array. Instead, 'divide and conquer', taking advantage of the fact that the array is sorted.

Min Of Sorted-Rotated

You will be given a numerical array that has first been sorted, then rotated by an unknown amount. Find and return the minimum value in that array.

Rotate Array

Implement **`rotateArr(arr, shiftBy)`** that accepts array and offset. Shift arr's values *to the right* by that amount. 'Wrap-around' any values that shift off array's end to the other side, so that no data is lost. Operate in-place: given **`([1,2,3],1)`**, change the array to **`[3,1,2]`**.

Optionally, add these advanced features:
a) allow a negative shiftBy (shift left, not right),
b) minimize memory usage. With only a few local variables (not an array), handle arrays and shiftBys in the millions,
c) minimize how many touches of each element.

Chapter 2 – Arrays

This chapter you will familiarize yourself with basic array manipulation. Here is a list of methods to study; some or all will be used in this chapter's challenges.

for / while loops *array.pop() & push()*
arrays grow: arr.length == lastIdx-1 *if / else statements*
can contain different data types *arrs are objects, passed by reference (ptr)*

Second-to-Last

Return the second-to-last element of an array.

Nth-to-Last

Return the element that is N-from-array's-end.

Second-Largest

Return the second-largest element of an array.

Nth-Largest

Given an array, return the Nth-largest element: there should be (N - 1) elements that are larger.

Chapter 2 – Arrays

Time-space Tradeoff

Good engineering is all about tradeoffs: knowing what tradeoffs are available, and knowing when to use them. In software engineering, one important tradeoff is *time vs. space*. If you know you will be asked to solve a certain formula repeatedly, you can keep track your previous answer and simply provide that answer rather than recomputing it. For certain problems, whether in algorithms class or in the workplace, *caching* (saving) the results does not make the function any faster when it is first called, but it can make subsequent calls *much* faster. Use this concept in today's algorithm challenges!

This chapter you will familiarize yourself with basic array manipulation. Here is a list of methods to study; some or all will be used in this chapter's challenges.

for / while loops *array.pop() & push()*
arrays grow: arr.length == lastIdx-1 *if / else statements*
can contain different data types *arrs are objects, passed by reference (ptr)*

arrConcat

Replicate JavaScript's **`concat()`**. Create a standalone function that accepts two arrays. Return a *new* array containing the first array's elements, followed by the second array's elements. Do not alter the original arrays. Ex.: **`arrConcat( ['a','b'], [1,2] )`** should return **`['a','b',1,2]`**.

Faster Factorial

Remember iFactorial from last chapter? Take that implementation and use a time-space tradeoff to accelerate the average running time. Recall that iFactorial(num) returns the product of positive integers from 1 to the given num. For example: **`fact(1)`** = 1, **`fact(2)`** = 2, **`fact(3)`** = 6. For these purposes, **`fact(0)`** = 1.

Shuffle

Recreate the **`shuffle()`** built into JavaScript, to efficiently shuffle a given array's values. Do you need to return anything from your function?

Chapter 2 – Arrays

This chapter you familiarized yourself with basic array manipulation. Some or all of these were used in this chapter's challenges.

for / while loops *array.pop() & push()*
arrays grow: arr.length == lastIdx-1 *if / else statements*
can contain different data types *arrs are objects, passed by reference (ptr)*

For extra array practice at the end of this chapter, work on these additional challenges:

Smarter Sum

Use a time-space tradeoff to accelerate the average running time of an iSigma(num) function that returns the sum of all positive integers from 1 to num. Recall: sig(1) = 1, sig(2) = 3, sig(3) = 6, sig(4) = 10.

Fabulous Fibonacci

Use a time-space tradeoff to accelerate the average running time of an iFibonacci(num) function that returns the 'num'th number in the Fibonacci sequence. Recall: fib(0) = 0, fib(1) = 1, fib(2) = 1, fib(3) = 2.

Tricky Tribonacci

Why stop with fibonacci? Create a function to retrieve a "tribonacci" number, from the sum of the previous 3. Tribonaccis are {0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, ...}. Again, use a time-space tradeoff to make this fast.

The “Buggy 13” (#1)

Below are solutions to the “Basic 13” algorithms. Unfortunately, some of these solutions contain errors. Which ones have bugs, and what are they?

Print1To255()

Print all the integers from 1 to 255.

```
function print1to255()
{
  var num = 1;
  while (num < 255) {
    console.log(num);
    num = num + 1;
  }
}
```

PrintIntsAndSum0To255()

Print integers from 0 to 255, and the sum so far.

```
function printSum0to255()
{
  var sum = 0;
  for(var num = 0;num <= 255;num++)
  {  sum += num;  }
  return sum;
}
```

PrintMaxOfArray(arr)

Print the largest element in a given array.

```
function printArrayMax(arr) {
  if (arr.length == 0) {
    console.log("[], no max val.");
    return;
  }
  var max = 0;
  for (var idx = 0;
       idx < arr.length; idx++) {
    if (arr[idx] > max)
    {  max = arr[idx];  }
  }
```

```
  console.log("Max value is:", max);
}
```

PrintOdds1To255()

Print all odd integers from 1 to 255.

```
function printOdds1to255()
{
  var num = 1;
  while (num <= 255)
  {
    console.log(num + 2);
  }
}
```

PrintArrayVals(arr)

Print all values in a given array.

```
function printArrayValues(arr) {
  for var idx = 0;idx < arr.length;
       arr++) {
    console.log("array[", idx,
                "] =",arr[idx]);
  }
}
```

PrintAverageOfArray(arr)

Analyze an array's values and print the average.

```
arr = [1,4,7,2,5,8];
if (arr.length == 0) {
  console.log("[ ], no avg val.");
  return;
}
var sum = arr[0];
for (var idx = 1; idx < arr.length;
     idx++) {
  sum += arr[idx];
}
console.log("Avg val:", sum/arr.length);
```

The “Bug-Ridden 13” (#1) – continued

(continued)

ReturnOddsArray1To255()

Create & return array with odd integers 1-255.

```
function oddArray1to255() {
  var oddArray = [];
  for (var num = 1; num <= 255;
       num += 2)
  { oddArray.push(num); }
}
```

ReturnArrayCountGreaterThanY(arr, y)

Given an array, return count greater than Y.

```
function numGreaterThanY(arr, y) {
  var numGreater = 0;
  for(var idx = 0; idx < arr.length;
      idx++) {
    if (arr[idx] > y)
    { numGreater++; }
  }
  return arr[y];
}
```

PrintMaxMinAverageArrayVals(arr)

Print the max, min and average array values.

```
function maxMinAverage(arr) {
  if (arr.length == 0) { return; }
  var min = arr[0];
  var max = arr[0];
  var sum = arr[0];
  for (var idx=1; idx <= arr.length;
       idx++) {
    if (arr[idx] < min)
    { min = arr[idx]; }
    if (arr[idx] > max)
    { max = arr[idx]; }
    sum += arr[idx];
  }
  return min;
```

```
  return max;
  return avg;
}
```

SquareArrayVals(arr)

Given an array, square each value in the array.

```
function squareArrVals(arr) {
  for (var idx = 0;idx < arr.length;
       idx++){
    arr[idx] = arr[idx] + arr[idx];
  }
}
```

ZeroOutArrayNegativeVals(arr)

Given an array, set negative values to zero.

```
setNegsToZero(arr) {
  for (var idx = 0;idx < arr.length;
       idx++){
    if (arr[idx] < 0){ arr[idx]=0; }
  }
}
```

ShiftArrayValsLeft(arr)

Shift array values forward, leaving `'0'` at end.

```
function arrShift(arr) {
  for (var idx = 1;idx < arr.length;
       idx++){
    arr[idx - 1] = arr[idx];
  }
  arr.length--;
  return arr;
}
```

SwapStringForArrayNegativeVals(arr)

Replace negative array values with `'Dojo'`.

```
function subStringForNegs(arr) {
  for (var idx = 0;idx < arr.length;idx++){
    if(idx < 0) {arr[idx]="Dojo"}
  }
  return arr;
}
```

Chapter 3 – Strings

This chapter explores associative arrays and then *strings* – a special case of the basic indexed array. By now you should be able to complete “Basic 13” algorithm challenges in less than 2 minutes each.

Remember best practices mentioned previously. Ask clarifying questions before rushing to write code. Think about and note any special-case situations (corner cases). Verify understanding by restating problems and intended outputs for simple input. *Then* start coding. Write T-diagrams to track variables.

Associative Arrays

Associative arrays (dictionaries, maps, key-value pairs) are JavaScript objects. Think of them as arrays indexed by strings. Associative arrays are initialized as {}, not [ ]. Access them as object attributes (**`arr.attrib`**) or array indices (**`arr["index"]`**).

Associative arrays / objects / maps / dictionaries

```
var myAssocArr = {};
myAssocArr.IQ = 116;
myAssocArr["fun"] = "Martin honks on a tenor saxophone";
console.log(myAssocArr);
      // { IQ: 116; fun: "Martin honks on a tenor saxophone" }
```

Arrs2Map

Given two arrays, create an associative array (map) containing keys of the first, and values of the second. For **`arr1 = ["abc", 3, "yo"]`** and **`arr2 = [42, "wassup", true]`**, return **`{"abc": 42, 3: "wassup", "yo": true}`**.

InvertHash

Create invertHash(assocArr) that converts a hash's keys to values and values to corresponding keys. Example: given `{"name": "Zaphod"; "numHeads": 2}`, return `{"Zaphod": "name"; 2: "numHeads"}`. You will need to learn and use a JavaScript `for ... in` here!

ReverseString

Implement a function reverseString(str) that, given a string, will return the string of the same length but with characters reversed. Example: given `"creature"`, return `"erutaerc"`. Do not use the built-in `reverse()` function!

Chapter 3 – Strings

Strings are arrays of characters (more accurately, arrays of one-character *strings*). Once a string is defined, individual characters can be referenced by [ ] but *not changed*. Strings are *immutable*: they can be completely replaced in their entirety, but not changed piecewise. To manipulate string characters, you must split the string to an *array*, make individual changes, then join it.

Strings

```
console.log(typeof myAssocArr.fun);    // "string"
var myChar = myAssocArr.fun[26];       // "x"
console.log(typeof myChar);            // "string"
```

.length method

```
console.log(myAssocArr.fun.length);    // 33
console.log("".length);                // 0
```

.split method

```
myArray = myAssocArr["fun"].split(" ");
      // ["Martin","honks","on","a","tenor","saxophone"]
console.log(myArray[5].split("");
      // ["s","a","x","o","p","h","o","n","e"];
```

.join method

```
console.log(myArray.join());
           // "Martin, honks, on, a, tenor, saxophone"
console.log(myArray.join("-"));
           // "Martin-honks-on-a-tenor-saxophone"
```

Challenge: what is displayed by the following? Why?

```
console.log(1 + 2 + "3" + "4" + 5 + 6);
```

Remove Blanks

Create a function that, given a string, returns the string, without blanks. Given `"  play  that Funky Music  "`, returns a string containing `"playthatFunkyMusic"`.

Get String Digits

Create a JavaScript function that given a string, returns the integer made from the string's digits. Given `"0s1a3y5w7h9a2t4?6!8?0"`, the function should return the number 1,357,924,680.

Acronyms

Create a function that, given a string, returns the string's acronym (first letters only, capitalized). Given `"there's no free lunch  -  gotta pay yer way"`, return `"TNFL-GPYW"`. Given `"Live from New York, it's Saturday Night!"`, return `"LFNYISN"`.

Chapter 3 – Strings

Switch/case statements

Think of switch statements as a series of if statements. From the **`switch(VAL)`**, execution jumps forward to the **`case:`** that matches the VAL (or **`default:`** if no match is found), continuing from there until an optional **`break;`**.

This chapter we explore associative arrays then strings. Some or all of these will be useful:

.length *.split* *.join* *.concat* *for...in loops* *switch/case*

Parens Valid

Create a function that, given an input string, returns a boolean whether parentheses in that string are valid. Given input **`"y(3(p)p(3)r)s"`**, return true. Given **`"n(0(p)3"`**, return **`false`**. Given **`"n)0(t(0)k"`**, return **`false`**.

Braces Valid

Given a string, returns whether the sequence of various parentheses, braces and brackets within it are valid. For example, given the input string **`"w(a{t}s[o(n{c}o)m]e)h[e{r}e]!"`**, return **`true`**. Given **`"d(i{a}l[t]o)n{e"`**, return **`false`**. Given **`"a(1)s[O(n]0{t)0}k"`**, return **`false`**.

Chapter 3 – Strings

This chapter we focus on *strings*. Some or all of these might be useful:

.length *.split* *.join* *.concat* *for...in loops* *switch/case*

Is Palindrome

Strings like **`"Able was I, ere I saw Elba"`** or **`"Madam, I'm Adam"`** could be considered *palindromes*, because (if we ignore spaces, punctuation and capitalization) the letters are the same from front and back.

Create a function that returns a boolean whether the string is a *strict* palindrome. For **`"a x a"`** or **`"racecar"`**, return **`true`**. Do **not** ignore spaces, punctuation and capitalization: if given **`"Dud"`** or **`"oho!"`**, return **`false`**.

Longest Palindrome

For this challenge, we will look not only at the entire string, but also substrings within it.

For a string, return the longest palindromic substring. Given **`"what up, dada?"`**, return **`"dad"`**. Given **`"not much"`**, return **`"n"`**. *Include spaces* as well (i.e. be strict, as in the "Is Palindrome" challenge): given **`"My favorite racecar erupted!"`**, return **`"e racecar e"`**.

Chapter 3 – Strings

Fast-finish / fast-fail

The idea of quickly exiting a function if a special case is detected likely does not seem all that revolutionary. However, this can not only simplify the code, but make its average running time faster as well. Whether to apply them to failure (fast-fail) or success cases will depend on the specifics of the challenge, but in any case they can quickly narrow a problem to the mainline case that remains.

This chapter's challenges focused on *maps / hashes*, then *strings*. Some or all of these concepts might be useful:

.length *.split* *.join* *.concat* *for...in loops* *switch/case*

Book Index

Write a function that given a sorted array of page numbers, return a string representing a book index. Combine consecutive pages to create ranges. Given `[1, 3, 4, 5, 7, 8, 10]`, return the string `"1, 3-5, 7-8, 10"`.

Common Suffix

When given an array of words, returns the largest suffix (word-end) that is common between all words. For example, for input `["ovation", "notation", "action"]`, return `"tion"`. Given `["nice", "ice", "sic"]`, return `""`.

Chapter 3 – Strings

This chapter you familiarized yourself with associative arrays and with strings. Here are some extra problems to keep you busy!

Why Algorithm Challenges Don't Allow Built-In Functions

Knowing available services for a language or framework is essential for unlocking its value. That said, there is power in knowing how to recreate those services if needed – if they don't work as expected, or when you must extend them for new scenarios. Furthermore, having a sense for how these services work 'under the hood' deepens your understanding about how/when to use them. Knowing for example that push() and pop() are *significantly* faster than splice() might make a difference in which you choose.

Recreating JavaScript String Library Functions

For extra algorithm practice, recreate these built-in functions from JavaScript's string library.

`string1.concat(string2,...,stringX)` - Add string(s) to end of existing one. Return new string.

`string.search(val)` - Search string for val. Return position of match (-1 if not found).
Bonus: hacker cred for implementing *regular expression* support!

`string.slice(start,end)` - Extract part of a string and return in a new one. *Start* and *end* are indices into the string, with the first character at index 0. *End* param is optional and if present, refers to one beyond the last character to include.
Bonus: include support for negative indices, representing offsets from string-end. Example: string.slice(-1) returns the string's last character.

`string.split(separator,limit)` - Split string into array of substrings, and return the new array. Separator specifies where to divide the substrings and is not included in any substring. If **`""`** is specified, split the string on every character. Limit is optional and indicates the number of splits; any additional items should not be included in the array. Note: existing string is unaffected.

`string.trim()` - Remove whitespace (spaces, tabs, newlines) from both sides, and return a new string. Example: **`" \n  hello goodbye \t ".trim()`** should return **`"hello goodbye"`**.

The “Bugful 13” (#2)

Below are solutions to “Basic 13”. Unfortunately, some contain errors. Which ones, and what are they?

Print1To255()

Print all the integers from 1 to 255.

```
function print1to255()
{
  for (var num = 1;num <= 255;num--)
  {
    console.log(num);
  }
}
```

PrintIntsAndSum0To255()

Print integers from 0 to 255, and the sum so far.

```
function printSum0to255()
{
  var sum = 0;
  for(var num = 0;num <= 255;num++){
    console.log(num,"  Sum:",sum);
    sum += num;
  }
}
```

PrintMaxOfArray(arr)

Print the largest element in a given array.

```
function printArrayMax(arr)
{
  if (arr.length == 0) {
    console.log("[ ], no max val.");
    return;
  }
  var max = arr[0];
  for (var idx = 1;idx < arr.length;
       idx++){
    if (arr[idx] > max) {
      max = arr[idx];
    }
```

```
    console.log("Max val:", max);
  }
}
```

PrintOdds1To255()

Print all odd integers from 1 to 255.

```
function printOdds1to255() {
  var num = 1;
  while (num <= 255) {
    console.log(num);
    num ++= 2;
  }
}
```

PrintArrayVals(arr)

Print all values in a given array.

```
function printArrayValues(arr)
{
  for(var idx=0; idx<arr.length;
      idx++) {
      console.log("array[", idx,
                  "] =", idx);
  }
}
```

PrintAverageOfArray(arr)

Analyze an array's values and print the average.

```
function printArrayAverage(arr)
{
  if (arr.length == 0) {
    console.log("[ ], no avg val.");
    return;
  }
  var sum = arr[0];
  for (var idx = 0;idx < arr.length;
       idx++)
  {sum += arr[idx];}
  console.log(sum/arr.length);
}
```

The “Bugger’s Banquet 13” (#2) – continued

ReturnOddsArray1To255()

Create & return an array with odd integers from 1-255.

```
function oddArray1to255() {
  // create empty array
  // setup for loop, with max iterations
  //   add # to array
  // return array
}
```

ReturnArrayCountGreaterThanY(arr, y)

Given an array, return the count that is greater than Y.

```
function numGreaterThanY(arr, y) {
  var numGreater = 0;
  for (var idx = 0;idx < arr.length;idx++){
    if (arr[idx] > y) { y++; }
  }
  console.log("%d values greater than %d", numGreater, y);
}
```

PrintMaxMinAverageArrayVals(arr)

Given an array, print max, min and average values.

```
function maxMinAverage(arr)
{
  if (arr.length == 0) {
    console.log("[] arr, no min/max/avg");
    return;
  }
  var min = arr[0];
  var max = arr[0];
  var sum = arr[0];
  for (var idx=1; idx <= arr.length;idx++){
    if (arr[idx] < min) { min = arr[idx]; }
    if (arr[idx] > max) { max = arr[idx]; }
    sum += arr[idx];
  }
  console.log("Max val:", max);
  console.log("Min val:", min);
  console.log("Avg val:", sum/arr.length);
```

```
  return [max, min, avg];
}
```

“Exterminator’s Delight” (#2) – continued

SquareArrayVals(arr)

Given an array, square each value in the array.

```
function squareArrVals(arr) {
  for (idx = 0; idx < arr.length; idx++) {
    arr[idx] = arr[idx * idx];
  }
}
```

ZeroOutArrayNegativeVals(arr)

Given an array, set negative values to zero.

```
function setNegsToZero(arr) {
  for (var idx = 1;idx < arr.length;idx++){
    if (arr[idx] < 0) {
      arr[idx] = 0;
    }
  }
}
```

ShiftArrayValsLeft(arr)

Given an array, shift values leftward by one. Drop first values and leave extra `'0'` value(s) at end.

```
function arrShift(arr) {
  for (var idx = 1;idx < arr.length;idx++) {
    arr[idx + 1] = arr[idx];
  }
  arr[arr.length - 1] = 0;
  return arr;
}
```

SwapStringForArrayNegativeVals(arr)

Given an array, replace negative values with `'Dojo'`.

```
function subStringForNegs(arr) {
  for (var idx = 0;idx < arr.length;idx++) {
    if (arr[idx] <= 0) {
      arr[idx] = "Dojo";
    }
  }
  return arr;
```

```
}
```

Chapter 4 – Linked Lists

This chapter we explore linked lists, a data structure used widely in lower layers such as backends, frameworks, runtimes or operating systems. You should be familiar with object oriented ideas, including the *reference* concept: not a local copy of a value, but a pointer to the value in shared memory.

How does your operating system keeps track of the files in a directory? Modern systems do not do this with an array. They use a data structure called a *linked list*. Linked lists are easily reordered and well-suited for large data collections because (unlike arrays) they store data in small pieces of memory that “fit in the holes” between variables, rather than requiring a large chunk of contiguous memory. Linked lists are the first data structure we discuss as an *object* and introduce us to the concept of a *reference*.

A **class definition** is like a blueprint of a complex machine, from which many copies can be made. Actually constructing a machine is a separate step. Likewise, *declaring* a class merely informs us of that blueprint; actual objects must be individually constructed. In JavaScript, class declarations take the form of functions called *object constructors* – when called, they create an **object** for the caller. An object is an instance of the class, brought to life, just like a physical copy of the ideas in the blueprint.

Not all machines are complex, and not all objects are complicated. However, code can add and remove attributes of objects on-the-fly, so this makes them different than a boolean or number which always occupies the same amount of memory space. Why does this matter? Well, if you have debugged your JavaScript code in the browser, you may understand the idea of a call stack. This is the series of function calls that led the computer to where it is right now. Whenever the currently running function returns, the JavaScript runtime will look to the call stack to help it “remember” which function it came from, as well as the state of all its local variables at the time when it called into another function. The runtime stores local variables in the call stack while changing the execution to another function. Setting aside call stack space for booleans and numbers is easy -- regardless of value, numbers occupy a 64-bit chunk of memory. However, objects are tricky: the JavaScript runtime cannot determine *a priori* how much space to set aside for your objects. So how can it quickly construct a call stack?

The answer is that objects are created using a common chunk of memory set aside for variable-sized allocations. This memory is called the *heap*, and it is used for any unpredictable memory needs. When the system looks at your ‘blueprint’ and constructs a ‘machine’ corresponding to those plans, it goes to the heap and sets aside space for all that object’s attributes and functions. If the object needs more space, it expands into adjacent heap memory. During normal operation, the heap is wide-open for large and small allocations. The call stack is apartment space in a high-rise tower; the heap is Montana.

When you create an object and store it in a local **`var`**, the system doesn’t put the object in that memory slot the way it does for a number or a boolean. It puts a *reference* to that heap location into your local **`var`**. References (called pointers) are fixed-size, so this enables the runtime do its stack magic. A pointer represents an object’s location in memory, but you can think of it as an object’s contact info: its

email address. True to its name, a pointer *points* to where the object is found. If you have information to retrieve from (or store to) an object, you “go there” by dereferencing that pointer, followed by the attribute you want within the object. This could look like **`myProject.name`** or **`myQuizzes[3]`** or even **`getAverage(myArr)`**. Yes, arrays, strings and even functions are objects – dereferenced by **`.`** or **`[`** or **`(`** .

Chapter 4 – Linked Lists

Over the chapter's course, we'll coalesce a considerable collection of concepts to contemplate. Some or all of these will be used in this chapter's challenges.

classes and *objects* *object constructors* *local vars vs. heap allocations* *pointers* *reference* vs. *value* *private* vs. *public* === vs. == *push( ) & pop( )*

To the left is a definition of a **node** object. A node object simply holds a value, as well as a *pointer* that links it to the next node in the sequence, if there is one. A sequence of node objects is called a *linked list*.

```
function listNode(value)
{
  this.val = value;
  this.next = null;
}
```

addFront

Given a pointer to the first node in a list, and a value, create a new node, connect it to the head of the list, and return a pointer to the list's new head node.

removeFront

Given a pointer to the first node in a list, remove the head node and return the new list. If list is empty, return null.

contains

Given a pointer to a listNode and a value, return whether value is found in any node within the list.

front

Return the *value* (not the node) at the head of the list. If list is empty, return null.

Chapter 4 – Linked Lists

This chapter we will familiarize ourselves with basic manipulation of the *singly linked list* data structure. Why is it referred to as a *singly* linked list? Well, there are many other ways to arrange node objects, and some of them feature more than one linkage between nodes. For example, *doubly linked list* nodes each connect to *two* others: the next one as well as the previous. *Singly* linked list nodes contain only a *next* pointer. Here are some of the concepts used in this chapter's challenges.

classes and *objects* *object constructors* *local variables vs. heap allocations* *push( ) & pop( )*
pointers *private* vs. *public* === vs. == *reference* vs. *value*

For the following challenges, use this listNode definition as a starting point. Note: *singly linked lists* are sometimes referred to as **sLists**.

length

Create a function that accepts a pointer to first list node, and returns number of nodes in sList.

average

Create a standalone function **`average(node)`** that returns (…wait for it …) the *average* of all values contained in that list.

```
function listNode(value)
{
  this.val = value;
  this.next = null;
}
```

min, max

Create function **`min(node)`** and **`max(node)`** to returning smallest and largest values in the list.

display

Create **`display(node)`** for debugging that returns a string containing all list values. Build what you wish **`console.log(myList)`** did!

Chapter 4 – Linked Lists

This chapter we familiarize ourselves with basic manipulation of the *singly linked list* data structure. Here are some concepts used in the chapter's challenges.

classes and *objects* *object constructors* *local variables vs. heap allocations*
pointers *private* vs. *public* === vs. == *reference* vs. *value*

As always, here's our node object:

```
function listNode(value) {
  this.val  = value;
  this.next = null;
}
```

back

Create a standalone function that accepts a listNode pointer and returns the last value in the linked list.

addBack

Create a function that creates a listNode with given value and inserts it at end of a linked list.

removeBack

Create a standalone function that removes the last listNode in the list and returns the new list.

Chapter 4 – Linked Lists

This chapter we familiarize ourselves with basic manipulation of the *singly linked list* data structure. Here are some concepts used in this chapter's challenges.

classes and *objects* *object constructors* *local variables vs. heap allocations*
pointers *private* vs. *public* === vs. == *reference* vs. *value*

Here is the humble-but-mighty listNode class:

```
function listNode(value)
{
  this.val  = value;
  this.next = null;
}
```

removeVal

Create **`removeVal(list,value)`** that removes from our list the node with the given **`value`**. Return the new list.

prependVal

Create **`prependVal(list,value,before)`** that inserts a listNode with given **`value`** immediately before the node with **`before`** (or at end). Return the new list.

appendVal

Create **`appendVal(list,value,after)`** that inserts a new listNode with given **`value`** immediately after the node containing **`after`** (or at end). Return the new list.

Chapter 4 – Linked Lists

This chapter you familiarized yourself with basic manipulation of the *singly linked list* data structure. Here are concepts used in this chapter's challenges.

classes and *objects* *object constructors* *local variables vs. heap allocations* *pointers* *private* vs. *public* === vs. == *reference* vs. *value*

Here's our listNode class:

```
function listNode(value) {
  this.val  = value;
  this.next = null;
}
```

splitOnVal

Create **`splitOnVal(list,num)`** that, given **`number`**, splits a list in two. The latter half of the list should be returned, starting with node containing **`num`**. E.g.: **`splitOnVal(5)`** for the list (1 >3>5>2>4) will change list to (1>3) and return value will be (5>2>4).

partition

Create **`partition(list,value)`** that locates the first node with that value, and moves all nodes with values *less than* that value to be earlier, and all nodes with values *greater than* that value to be later. Otherwise, original order need not be perfectly preserved.

deleteGivenNode

Create *listNode method* **`removeSelf()`** to disconnect (remove) itself from linked lists that include it. Note: the node might be the first in a list, and you do NOT have a pointer to the previous node. Also, don't lose any subsequent nodes pointed to by **`.next`**.

Chapter 4 – Linked Lists

This chapter you familiarized yourself with basic manipulation of the *singly linked list* data structure. Here are concepts used in this chapter's challenges.

classes and *objects* *object constructors* *local variables vs. heap allocations*
pointers *private* vs. *public* === vs. == *reference* vs. *value* *time vs. space tradeoff*

So far, here is what we've created for the linkedList and listNode classes:

```
function listNode(value) {
  this.val  = value;
  this.next = null;
  this.removeSelf = function() { ... }
}
```

dedupeList

Remove nodes with duplicate values. Following this call, all nodes remaining in the list should have unique values. Retain only the first instance of each value.

dedupeListWithoutBuffer

Can you accomplish this without using a secondary buffer? What are the performance ramifications?

The “Bug-Laden 13” (#3)

Below are “Basic 13” solutions. Unfortunately, some contain errors. Which ones, and what are they?

Print1To255()

Print all the integers from 1 to 255.

```
function print1to255()
{
  var num = 1;
  while (num <= 255) {
    console.log(num);
  }
}
```

PrintIntsAndSum0To255()

Print ints 0-255, and with each the sum so far.

```
var sum = 0;
function printSum0to255()
{
  for (var num = 0;num <= 255;num++)
  {
    sum += num;
    console.log(num," – sum:",sum);
  }
}
```

PrintMaxOfArray(arr)

Print the largest element in a given array.

```
function printArrayMax(arr)
{
  var max = arr[0];
  for(var idx = 1;idx < arr.length;
      idx++){
    if (arr[idx] > max) {
      max = arr[idx];
    }
  }
  console.log("Max value is:", max);
```

```
}
```

PrintOdds1To255()

Print all odd integers from 1 to 255.

```
function printOdds1to255() {
  var num = 1;
  while (num <= 255) {
    num = num + 2;
    console.log(num);
  }
}
```

PrintArrayVals(arr)

Print all values in a given array.

```
function printArrayValues()
{
  for(var idx=0;idx<arr.length;idx++){
    console.log("array[", idx, "] equals",arr[idx]);
  }
}
```

PrintAverageOfArray(arr)

Analyze an array's values; print the average.

```
function printArrayAverage(arr)
{
  if (arr.length == 0) {
    console.log("[], no average.");
    return;
  }

  var sum = arr[0];
  for(var idx = 1;idx < arr.length;
      idx++) {
    sum += arr[idx];
    console.log(sum/arr.length);
  }
}
```

The “Bug-alicious” 13 (#3) – continued

ReturnOddsArray1To255()

Create & return array with odd integers from 1-255.

```
function oddArray1to255() {
  var oddArray = [];
  for (var num = 1; num <= 255; num += 2) {
    var temp = num;
    oddArray.push(temp);
  }
  return oddArray;
}
```

ReturnArrayCountGreaterThanY(arr, y)

Given array, return the count that is greater than Y.

```
var numGreater = 0;
function numGreaterThanY(arr, y) {
  for (var idx = 0; idx < arr.length; idx++) {
    if (arr[idx]>y) { numGreater ++; }
  }
  console.log("%d are larger than %d", numGreater, y);
}
```

PrintMaxMinAverageArrayVals(arr)

Given an array, print max, min and average values.

```
function maxMinAverage(arr) {
  if (arr.length == 0) {
    console.log("[], no min/max/avg");
    return;
  }
  var min = arr[0], max = arr[0];
  var sum = arr[0];
  for(var idx = 1; idx <= arr.length; idx++) {
    if (arr[idx] < min) { min = arr[idx]; }
    if (arr[idx] > max) { max = arr[idx]; }
    sum += arr[idx];
  }
  console.log("Max value:", max);
  console.log("Min value:", min);
  console.log("Avg:",sum/arr.length);
```

```
  return min, max, avg;
}
```

“Forrest and His Friend Bugga Gump” (#3) – continued

SquareArrayVals(arr)

Given an array, square each value in the array.

```
function squareArrVals(arr)
{
  var numSquareValues;
  for (var idx=0;idx<arr.length;idx++) {
    arr[idx] = arr[idx] * arr[idx];
  }
  return numSquareValues;
}
```

ZeroOutArrayNegativeVals(arr)

Given an array, set negative values to zero.

```
function setNegsToZero() {
  for (var idx=0;idx<arr.length;idx++) {
    if (arr[idx]<0) { arr[idx] = 0; }
  }
}
```

ShiftArrayValsLeft(arr)

Given array, shift values forward by one, drop the first values and leave extra `'0'` value(s) at the end.

```
function arrShift(arr) {
  for (var idx=1;idx<arr.length;idx++)
  { arr[idx - 1] = idx; }
  arr[arr.length - 1] = 0;
  return arr;
}
```

SwapStringForArrayNegativeVals(arr)

Given array, replace negative values with `'Dojo'`.

```
function subStringForNegs(arr) {
  for(var idx=0;idx<arr.length;idx++) {
    if (arr[idx] < 0)
      arr[idx] == "Dojo";
    }
  }
  return arr;
```

```
}
```

Chapter 5 – Queues and Stacks

Hey! Hold on to this book for me. Also, could you take this note-to-self I wrote on a slip of paper? Oh, and I almost forgot – someone important called for you. Here's the phone number – ready?

It's crazy how much we are asked to commit to, and recall from, memory on a daily basis. That's why we have machines save this information for us! Devices are better than humans at storing data *primarily* we are ***faster*** (and not forgetful!) and have an expandable amount of storage space.

When we create software systems, we make choices about how we store and organize information, and these choices significantly impact the performance of our systems. Over time, well-known patterns have emerged for storing, managing and retrieving information. These patterns are reflected in reusable code called *data structures*.

Put simply (and obviously), data structures handle *data* - they store, organize, and retrieve information. There are many different data structures; each exists because it is optimized for a certain set of usage scenarios. Said another way, each data structure has its own priorities about what aspects are important. As a result each data structure has strengths and weaknesses that make it well-suited for certain situations and ill-suited for others.

In general, these choices are design tradeoffs that data structure creators make: how the data structure consumes memory, which of its functions it expects to be most frequently called, etc. Understanding these tradeoffs enables you to make intentional decisions about which data structure to use in your particular situation. If you know, for example, that you will constantly search your data structure for random values, but will rarely add or remove values from it, then you might choose a BST for your data structure, instead of a linked list.

You have already worked closely with a number of data structures previously – these include singly and doubly linked lists, arrays, and binary (search) trees. This chapter we will learn about a number of new data structures, including the Stack and various flavors of Queues. We will also dive into how existing data structures change when we change how they deal with duplicate values. As we study data structures, it is important to keep in mind that these data structures could be implemented in a number of ways, using different building blocks underneath. For this reason, data structures such as Queues are considered *Abstract Data Types*. They are considered *abstract* because the outward behavior of the data structure is well understood, but there is no requirement on how the data structure is constructed internally. We could choose to rewrite an existing Abstract Data Type, and as long as we maintain the same Abstract interface, there should be no problems.

Chapter 5 – Queues and Stacks

Queues

Ever since we were young, we were taught to *wait our turn*. The Queue data structure enforces this notion and is considered a *Sequenced* data structure. The order in which data values are added is the order in which they exit the data structure. Just like when we wait in line at the ice cream store, the same when we add elements to a Queue: the first value added to the queue will be first to emerge as we start retrieving elements from or queue (likewise, the first customer to arrive and wait in line is the first to get a tasty treat!). For this reason, the Queue interface contains few methods. These include:

`enqueue(val)`: add val to queue
`dequeue()`: remove & return front value
`front()`: return (not remove) front value
`contains(val)`: does queue contain val?
`isEmpty()`: does queue contain nothing?
`size()`: return number of vals in queue

As with any Abstract Data Structure, we can implement a *Queue* in multiple ways. Today, we will create a Queue using an underlying *singly linked list*. Below is our reference. Let's do it!

```
function node(value) {
  if (!(this instanceof node)) {
    return new node(value);
  }
  this.val  = value;
  this.next = null;
}
```

```
function slQueue() {
  if (!(this instanceof slQueue)) {
    return new slQueue();
  }
  var front = null;
  var back  = null;
}
```

Enqueue

Create **`slQueue`** method **`enqueue(val)`** to add the given value to end of our queue. Remember, **`slQueue`** uses a singly linked list (not an array).

Dequeue

Create **`slQueue`** method **`dequeue()`** to remove and return the value at front of queue. Remember, **`slQueue`** uses a singly linked list (not array).

Front

Create **`slQueue`** method **`front()`** to return the value at front of our queue, without removing it.

Contains

Create method **`contains(val)`** to return whether given value is found within our queue.

Is Empty

Create **`slQueue`** method **`isEmpty()`** that returns whether our queue contains no values.

Size

Create **`slQueue`** method **`size()`** that returns the number of values in our queue.

Chapter 5 – Queues and Stacks

Stacks

Stacks and Queues are companion data structures. Both are *sequential* data structures, meaning that they manage their data according to the order in which they were added. Whereas a Queue data structure operates by the principle of “First-In becomes First-Out” (FIFO), a Stack is quite the opposite (Last-In, First-Out or LIFO).

Consider a pile of papers. With this stack of papers, we can only get a good look at the paper on the top of the pile. When we add another paper to the stack, *that* page becomes the only one visible. We can only add and remove papers from the top of the pile. We do not have the ability to add a page to the middle of the stack (just as someone should not cut into the middle of the *queue* at the ice cream store). In this way, Stacks and Queues mirror one another. Their basic methods correspond perfectly: simply substitute **push / pop / top** for **enqueue / dequeue / front**, and they become identical.

Stack Implementation Based on Array

Stacks can be easily implemented with arrays. How rapidly can you build the essential six methods (**`push`**, **`pop`**, **`top`**, **`contains`**, **`isEmpty`**, **`size`**) for a Stack class **`arrStack`** using an underlying *array*?

OK, good! Now that you’re warmed up, create a list-based class **`slStack`**, with a *singly linked list*:

Stack Push

Create **`push(val)`** that adds val to our stack.

Stack Pop

Create **`pop()`** to remove and return the top val.

Stack Top

Return (not remove) the stack’s top value.

Stack Contains

Return whether given val is within the stack.

Stack Is Empty

Return whether the stack is empty.

Stack Size

Return the number of stacked values.

Chapter 5 – Queues and Stacks

In the code you've written the past few days, you may have seen the significant similarity between Queues and Stacks. Today we will use that similarity to our advantage, reducing our code *footprint*.

Stack / Queue Code-Sharing

As a design exercise, think through how you would combine the **`slQueue`** you wrote previously with the **`slStack`** class you just created. Would you use object-oriented design? If so, which class inherits from which? Is there a parent class that is neither?

Once you've done the thought work, now it is time to code it: rework what you wrote for **`slQueue`** and **`slStack`** (or start from scratch, calling these **`slQueue2`** and **`slStack2`**) with this code-sharing in mind. When you are done, someone should still be able to create a new queue object or a new stack object and use any of the methods for that class, as before. However, the combined codebase should be only about 15% larger than **`slQueue`**!

Deque Class Implementation

Having combined the Stack and Queue classes *designwise*, why not combine them *featurewise* as well. Create a class called Deque (pronounced 'deck') that represents a *double-ended queue*. In addition to the basic six methods, add versions that push and pop from the opposite ends. Specifically, create a class **`deque`**, containing methods **`pushFront(val)`**, **`pushBack(val)`**, **`popFront()`**, **`popBack()`**, **`front()`**, **`back()`**, **`contains(val)`**, **`isEmpty()`**, and **`size()`**.

Chapter 5 – Queues and Stacks

Because of the one-way direction of singly linked lists, using a list to implement a Queue feels natural. What if we wanted to use an underlying *Array*? Why would we do this? In high-performance scenarios, when working with our queue we may not be able to allocate memory to create new node objects. Although we could allocate a large number of empty node objects ahead of time and keep them in a resource pool, what if instead we built a Queue class *using an array* as the underlying data structure?

Arrays are a natural choice for a *Stack data structure*, since Stacks add and remove from the same end, just like Stacks do. They even both have push() and pop() methods. To use an Array underneath a *Queue*, however, there is a wrinkle – at least after a while. With both Queue and Stack, as we add elements our Array gets longer, since elements are placed at the end of the Array. With a Stack, as we remove elements our Array grows and shrinks back like an accordion, since they are removed from the end (the [0] side of the Array never changes). This isn't the case with a Queue: we add elements to the end, but we remove them from the beginning. We will need to track the head index and the tail index. Unfortunately (and here is the wrinkle), over time as elements are added and removed, our array will get very large, as our head and tail indices grow higher and higher. This eats up memory even worse than allocating (and freeing!) list node objects. What to do? Start by capping our Array's size!

Circular Queues

When Queue's tail or head approaches 'size', wrap around to [0] and continue. We cannot let tail and head meet – one can't "lap" the other. Instead, enqueue(val) should *fail*: Queue is *full*. Ditto dequeue() if Queue is *empty*. Constructor requires a size argument. Starting there, let's create a circular queue implementation!

Enqueue

Create enqueue(val) that adds val to our circular queue. Return false on fail. Wrap if needed!

Front

Return (not remove) the queue's front value.

Is Empty

Return whether queue is empty.

Is Full

Return whether queue is full.

```
function cirQueue(cap) {
  if (!(this instanceof cirQueue))
  { return new cirQueue(cap); }
  var head = 0;
  var tail = 0;
  var capacity = cap;
  var arr = [];
}
```

Dequeue

Create cirQueue method dequeue() that removes and returns the front value. Return null on fail.

Contains

Return whether given val is within the queue.

Size

Return number of queued vals (not capacity).

Grow

(advanced) Create method grow(newSize) that expands the circular queue to a new given size.

The “Bug-Infested 13” (#4)

Below are “Basic 13” solutions. Unfortunately, some contain errors. Which ones, and what are they?

Print1To255()

Print all the integers from 1 to 255.

```
function print1to255() {
  for (var num = 1;num <= 255;num++) {
    console.log(num);
}
```

PrintIntsAndSumTo255()

Print integers from 0 to 255, and the sum so far.

```
function printSum0to255()
{
  var sum = 0;
  for (var num=0; num<=255;
       sum+=num,num++) {
    console.log(num, " - sum:", sum);
```

```
  }
}
```

PrintMaxOfArray(arr)

Print the largest element in a given array.

```
function printArrayMax(arr)
{
  if (arr.length == 0) {
    console.log("[ ], no max val.");
  }
  var max = arr[0];
  for (var idx=1; idx < arr.length;
      idx++)  {
    if (arr[idx] > max)
    { max = arr[idx]; }
  }
  console.log("Max val:", max);
}
```

PrintOdds1To255()

Print all odd integers from 1 to 255.

```
function printOdds1to255() {
  for (var num = 1;num <= 255;num+=2)
  { print(num); }
}
```

PrintArrayVals(arr)

Print all values in a given array.

```
function printArrayValues(arr)
{
  for (var idx=0; idx < arr.length;
       idx++)
    console.log("arr[",index,"]=",arr);
}
```

PrintAverageOfArray(arr)

Analyze an array's values and print the average.

```
function printArrayAverage(arr)
{
  if (arr.length == 0) {
    console.log("Empty arr, no avg val.");
    return;
  }

  var sum = arr[0];
  for(var idx=1; idx<arr.length-1;
      idx++) {
    sum += arr[idx];
  }
  console.log("Avg:",sum/arr.length);
}
```

“Ain’t Too Proud to Bug” (#4) – continued

ReturnOddsArray1To255()

Create & return array with odd integers 1-255.

```
function oddArray1to255(arr) {
  var oddArray = [];
  for (var num=1; num<=255; num+=2) {
    oddArray.push(num);
  }
  return oddArray;
}
```

ReturnArrayCountGreaterThanY(arr, y)

Given array, return count greater than Y.

```
function numGreaterThanY(arr, y) {
  var numGtr;
  for (var idx=0; idx<arr.length;
       idx++) {
    if (arr[idx] > y) { numGtr ++;}
  }
  console.log("%d > %d",numGtr, y);
  return numGtr;
}
```

SquareArrayVals(arr)

Given an array, square each value in the array.

```
function squareArrVals(arr)
{
  for (var idx = 0;idx < arr.length)
       idx++)
  {
    arr[idx] = arr[idx] * arr[idx];
  }
}
```

ZeroOutArrayNegativeVals(arr)

Given an array, set negative values to zero.

```
function setNegsToZero(arr)
{
  for(var idx=0;idx<arr.length;idx++){
    if (idx < 0)
    { arr[idx] = 0; }
  }
}
```

ShiftArrayValsLeft(arr)

Given array, shift values forward. Drop first value(s) and leave extra `'0'` value(s) at end.

```
function arrShift(arr)
{
  for (var idx = 1;idx < arr.length;
       idx++) {
    arr[idx - 1] = arr[idx];
  }
  return arr;
}
```

SwapStringForArrayNegativeVals(arr)

Replace negative array values with `'Dojo'`.

```
function subStringForNegs(arr) {
  for (var idx=0;idx < arr.length;
       idx++){
    if (arr[idx] < 0) {
      arr[idx] = "Dojo";
    }
  }
  return arr;
}
```

“Bilbo Buggins and the Unexpected Defects” (#4) – continued

PrintMaxMinAverageArrayVals(arr)

rint max, min and average array values.

```
function maxMinAverage(arr) {
  if (!arr.length) {
    console.log("[], no workie!");
    return;
  }
  var min, max, sum;
  min = max = sum = arr[0];
  for(var idx=1; idx<arr.length;
      idx++) {
    if(arr[idx]<min) {min=arr[idx];}
  }
  for(var idx=1; idx<arr.length;
      idx++) {
    if(arr[idx]>max) {max=arr[idx]; }
  }
  for(var idx=1; idx<arr.length;
      idx++) { sum += arr[idx]; }
  console.log("Max val:", max);
  console.log("Min val:", min);
  console.log(sum/arr.length);

function maxMinAverage(arr) {
  if (arr.length == 0) {
    console.log("[], no min/max/avg");
    return;
  }
  var min = arr[0];
  var max = arr[0];
  var sum = arr[0];
  for(var idx=1;idx<=arr.length;idx++){
    if (arr[idx]<min) { min=arr[idx]; }
    if (arr[idx]>max) { max=arr[idx]; }
    sum += arr[idx];
  }
  console.log("Max value:", max);
  console.log("Min value:", min);
}
```

```
function maxMinAverage(a)
{
  if (a.length == 0)
  {
   console.log("[],no min/max/avg");
   return;
  }
  var min,max,sum;
  min = max = sum = a[0];
  for(var idx=1;idx<a.length;idx++){
    if(a[idx]<min)     {min=a[idx];}
    else if(a[idx]>max){max=a[idx];}
    else               {sum+=a[idx];}
  }
  console.log("Max:",max);
  console.log("Min:",min);
  console.log("Avg:",sum/a.length);
}
```

Chapter 6 – Arrays, Part II

Interview Tips

Students sometimes wonder *what language* they should use in technical interviews. The answer of course is "whatever language you are told to use." What if you don't know what language they prefer? Or what if you *do* know, but you are not strong in that language? Don't despair! Most interviewers say "*Write in whatever language makes you most comfortable….*" Keep in mind they also think "(*...as long as it is an appropriate choice for this problem*)" so don't choose Ruby when interviewing to write graphics firmware; don't choose Fortran for a front-end interview. Except for highly specialized roles, it is safe to write in JavaScript, C++ or Java. As you know, we focus on JavaScript in this algorithm course, as it is universal to all web front-ends, and has growing server-side usage with Node.js.

Interviewing is an artificial situation, but like anything else you can improve your performance with practice. Even after the bootcamp, *practice coding on whiteboard* and on paper, to simulate interviews. Resist the urge to practice only at the computer. Once you have completed a solution on whiteboard or paper, *then* enter it into the computer and debug what you wrote. What common errors do you make when coding on whiteboard or paper? Make note of these and refer back to them from time to time.

How and what you *communicate to the interviewer* can be as important as getting the right solution to a problem. Remember that s/he can't read your mind, though, so you must always *think out loud*. Even if you are going through multiple possibilities mentally, discarding numerous deadend ideas, it still benefits you to give the interviewer visibility into your thought process.

Don't jump in and start writing code immediately. Ask *clarifying questions* - the answers might surprise you. Often, interview challenges are intentionally described in vague terms, to test whether a candidate can extract unstated requirements. Ensuring you understand the intention upfront is important. Asking about extreme inputs, including those that violate the function's expectations (whether or not the caller did this intentionally) is almost always useful -- jot these on a corner of the whiteboard to double-check later. Likewise, it is valuable to restate back to the interviewer your understanding of the problem. List on the board a few example inputs, along with expected outputs. If you can think of multiple ways to solve the problem (and perhaps even if not), it is almost always enlightening to ask "What are we optimizing for?" Really listen to the interviewer's responses in all of these.

Sooner or later you do have to start coding – don't let the pre-coding phase stretch out too long. A common-sense tip: start coding in the upper left corner of the whiteboard. Leave yourself room so you don't need a bunch of arrows - engineers that plan ahead are better engineers…. Start with the function's signature (name and inputs) and the first few lines of input error-checking, just to "get some ink on the board". If in the middle of the function you get stuck, don't stall - leave a comment or a bit of pseudocode and mention to the interviewer that you will come back to this. Keep going; try to maintain velocity. Mid-stream, you may realize there is a much better solution. Don't keep this a secret; your

interviewer already knows this. Mention it but suggest you keep going in order to finish something in the time you have. Always keep your ears open for the interviewer’s hints / guidance.

Chapter 6 – Arrays, Part II

This chapter covers additional *array* challenges. As we work with this data structure, we grow confidence and velocity in loops & conditionals. Now, if needed, we can use newer concepts like recursion. Put yourself into the mindset of a technical interview during this chapter's challenges, using the following:

Don't panic *think out loud* *clarifying questions*
error and corner cases *example inputs* *diagrams*
admit when it is suboptimal (but keep going) *"what are we optimizing for?"*

Average

(Warmup) Return average value of a given array.

Balance Point

Write a function that returns whether the given array has a balance point between indices, where one side's sum is equal to the other's. Example: `[1,2,3,4,10]` → `true` (*between indices 3 & 4*), but `[1,2,4,2,1]` → `false`.

Balance Index

Here, a balance point is *on* an index, not *between* indices. Return the *balance index* where sums are equal on either side (exclude its own value). Return -1 if none exist. Ex.: `[-2,5,7,0,3]` → `2`, but `[9,9]` → `-1`.

Taco Truck

You drive a taco truck; an array of [x,y] coordinates corresponds to the locations of your customers. You need to minimize the *total distance* from truck to customers. The truck only parks on street corners (at integer coordinates like [37,-16]; customers only travel on streets (coordinate [2,-2] is distance 4 from [0,0]). Given the customer array, determine the optimal place for taco truck to park.

Chapter 6 – Arrays, Part II

During these Array challenges, put yourself into an interview mindset, using the following concepts:

Don't panic think out loud clarifying questions error and corner cases example inputs diagrams admit when it is suboptimal (but keep going) "what are we optimizing for?"

Flatten

Flatten a given array, eliminating nested and empty arrays. Do not alter the array; return a new one retaining the order. Given the array `[1,[2,3],4,[]]`, return a new `[1,2,3,4]`.

Second: Work *'in-place'* in the given array (do not create another). Alter order if needed. Ex.: `[1,[2,3],4,[]]` could become `[1,3,4,2]`.

Third: Make your algorithm both *in-place* and *stable*. Do you need a return value?

Mode of Array

Back in the Basic 13, you wrote code to compute an array's minimum and maximum values. You also wrote code to determine average value (the "mean"). What about the "mode" – the most common value in that data set. Create a function that, given an array, returns the most frequent value in the array.

Second: if because of memory constraints you were forced to do this without creating a new array, how would it affect your solution?

Remove Duplicates

Remove duplicates from an array. Do not alter the original array; return a new one, keeping results *'stable'* (retain original order). Given [1, 2, 1, 3, 4, 2], return a new array [1, 2, 3, 4].

Second: Work *'in-place'* in given array. Alter order if needed (*stability* is not required). Ex.: `[1,2,1,3,4,2]` could become `[1,2,4,3]`.

Third: Make it *in-place* and *stable*.

Fourth: Can you make this faster by eliminating any second inner loop?

Chapter 6 – Arrays, Part II

This chapter we dive deeply into Arrays. Put yourself into the mindset of a technical interview during this chapter's algorithm challenges, using the following concepts:

Don't panic think out loud clarifying questions error and corner cases example inputs diagrams admit when its suboptimal (but keep going) "what are we optimizing for?"

Median of Sorted Arrays

Given two arrays that are sorted but not necessarily the same length, find the median value. For example, if given `([1,5,9], [1,2,3,4,5,6])`, return `4`. If the number of values is even, return the average of the two middle values. if given `([1,5,9], [1,2,3,4,5])`, return `3.5`.

Second: Expand your function to accept three arrays instead of two.
Third: Rework your function to correctly handle an arbitrary number of arrays.

Time to English

You are given an integer representing the number of minutes that have elapsed since midnight. You should return a string representing the current time, in traditional spoken convention. Use numerals, except specifically the following words – *midnight*, *noon*, *past*, *til*, *half*, *quarter*. Examples: if given `30`, return `"half past midnight"`. If given `75`, return `"quarter past 1 am"`. If given `710`, return `"10 til noon"`. If given `1000`, return `"20 til 5 pm"`.

Rain Terraces

The Seattle Coding Dojo wants to send excess water to the Burbank Coding Dojo, so it landscapes its rooftop with a set of unusual elevated terraces. They are all the same width, but have varying heights. When it rains, water gathers in low terraces that are surrounded by taller ones. For example, if we have terraces with heights `[3,1,1,4,2]`, then as much as 4 units of water could be gathered, because water would pool 2-deep on two different terraces (both of the 1-high terraces: between the 3-high and 4-high terraces). Water on the other terraces just runs off. Given an array of terrace heights, return the maximum amount of water that is trapped when rains come.

Chapter 6 – Arrays, Part II

This chapter we dove deeply into Arrays. Put yourself into the mindset of a technical interview during this chapter's algorithm challenges, using the following concepts:

Don't panic *think out loud* *clarifying questions* *error and corner cases* *example inputs*

Matrix Search

You will be given 2 different two-dimensional arrays, containing integers between 0 and 65535. Each two-dimensional array represents a black-and-white image, where each integer value is a pixel value. The second matrix might be a subset of the larger one. Return whether the second image is found within the larger one.

Below is an end-of-chapter challenge. How efficient can you be? The input may contain many million values.

Max of Subarray Sums

Given a numerical array that is potentially very long, return the maximum sum of values from a subarray. Any *consecutive sequence* of indices in the array is considered a subarray. Create a function that returns the highest sum possible from these subarrays. Given `[1,2,-4,3,-2,3,-1]`, you should return **4** (for subarray `[3,-2,3]`), and given `[-1,-2,-4,-3,-2,-3]`, return **0** (for `[]`). This problem has many possible implementations. Which do you prefer & why?

Chapter 7 – Linked Lists, Part II

Now we revisit *linked list* and *object pointer manipulation*. Later we will see a list type where each node connects to *two* others, but in our original linked list, each node links only to the *next*. It is accurately called a *singly linked list*, where iterating forward is easy; backward is tricky. Start from this sList object:

```
function slNode(value) {
  this.val = value;
  this.next = null;
}
function sList() {
  this.head = null;
  this.back = function() {
    if (!this.head) { return null; }
    var runner = this.head;
    while (runner.next)
    { runner = runner.next; }
    return runner.val;
  }
  this.pushBack = function(value) {
    var newNode = new slNode(value);
    if (!this.head)
    { this.head = newNode; }
    else {
      var runner = this.head;
      while (runner.next)
      { runner = runner.next; }
      runner.next = newNode;
    }
  }
  this.popBack = function() {
    if (!this.head) { return null; }
    var returnVal;
    if (!this.head.next) {
      returnVal = this.head.val;
      this.head = null;
      return returnVal;
    }
    var runner = this.head;
    while (runner.next.next)
    { runner = runner.next; }
    returnVal = runner.next.val;
```

```
    runner.next = null;
    return returnVal;
  }
```

```
  this.pushFront = function(value) {
    var oldHead = this.head;
    this.head = new slNode(value);
    this.head.next = oldHead;
  }
  this.popFront = function() {
    var returnVal = null;
    if (this.head) {
      returnVal = this.head.val;
      this.head = this.head.next;
    }
    return returnVal;
  }
  this.contains = function(value) {
    var runner = this.head;
    while (runner) {
      if (runner.val === value)
      { return true; }
      runner = runner.next;
    }
    return false;
  }
  this.removeVal = function(value) {
    if (!this.head) { return false;}
    if (this.head.val === value) {
      this.head = this.head.next;
      return true;
    }
    var runner = this.head;
    while (runner.next) {
      if (runner.next.val===value) {
       runner.next=runner.next.next;
       return true;
      }
      runner = runner.next;
    }
    return false;
  }
}
```

Chapter 7 – Linked Lists, Part II

Over the chapter's course, we'll coalesce a considerable collection of cool concepts for contemplation:

classes and *objects* *private* vs. *public* *prototype* === vs. == *reference* vs. *value*

This chapter's challenges have an *optional additional twist, if you so choose*. If the challenge asks for a method in the *sList* class, then to accept this optional challenge, solve it three ways: as a standalone function that accepts an *sList* object, as an *slNode* method, and as an *sList* method.

Reverse

Reverse the sequence of nodes in the list.

KthLast

Given k, return the value that is 'k' nodes from the list's end. If given `(list,1)`, return the list's last value. If given `(list,4)`, return the value at the node that has exactly 3 nodes following it.

Is Palindrome

Return whether a list is a palindrome. *String* palindromes read the same front-to-back and back-to-front. Here, compare *node values*. N.B.: to be accurate in JavaScript, use **===** instead of **==**, since `1 == true == [1] == "1"`.

Second: depending on environment, you might not have plentiful memory available. Can you solve this without using an additional array?

Chapter 7 – Linked Lists, Part II

Be mindful of these ideas as you work through the chapter's challenges:

classes and *objects* *private* vs. *public* *prototype* === vs. == *reference* vs. *value*

Today's challenges should all be implemented as *standalone functions*.

Sum List Numerals

You are given two lists, each representing a number. Every node value is a 0-9 digit, with *first* node representing *least significant digit*. Return a *new* list representing the sum. Given 2=>0=>1 & 8=>4, return 0=>5=1 because 102 + 48 = 150.

Second: what if first node is **most** significant?

Flatten Child Lists

Why limit nodes to contain only one pointer? For this challenge, each node has `.next` as always, but also a `.child` that is either `null` or points to the head of another list. Each node in those *child* lists might point to another list, and so on. Don't alter `.child`, but rearrange `.next` pointers to 'flatten' this hierarchy into a one linear list, from head node through all others via `.next`.

Setup List Loop

In preparation for tomorrow, create a sequence of `slNode`s that form a closed loop. Your function's first argument should signify how many nodes total, and the second should be which node number is pointed to by the last node. Give nodes sequential numbers as values, for clarity. Callling setupLoop(5, 3) should return a circular list of 1=>2=>3=>4=>5=>3=>4=>5=>3....

Shift List

Given a list, shift nodes to the right, by a given number *shiftBy*. These shifts are circular: i.e. when shifting a node off list's end, it should reappear at list's start. For list (a)=>(b)=>(c), shift(1) should return (c)=>(a)=>(b).

Second: also handle negative *shiftBy* (to left).

Unflatten Child Lists

Take the output from your "flatten child lists" function (a linear linked list containing nodes with **`.child`** pointers), and restore it to its original state. Do you need to change your *flatten* function to enable this?

Second: for each (**`flatten`** & **`unflatten`**), what is your big-O? Can you make these O(n)?

Chapter 7 – Linked Lists, Part II

Be mindful of these ideas as you work through the chapter's challenges:

classes and *objects* *private* vs. *public* *prototype* === vs. == *reference* vs. *value*

Build standalone functions and *sList* & *slNode* methods. Previous **`setupListLoop()`** may be useful.

Has Loop

Given a linked list, determine whether it has a loop, and return a boolean accordingly.

Loop Start

Given a linked list, return a pointer to the node where loop begins (where last node points), or null if no loop.

Break Loop

Given a linked list, determine whether there is a loop, and if so, break it. Retain all nodes, in original order.

Number of Nodes

Given a linked list with or without a loop, return total number of nodes. Given circular list (a)=>(b)=>(c)=>(d)=>(e)=>(c), return 5.

Chapter 7 – Linked Lists, Part II

Where's the Bug? (sList version)

Without peeking at previous code, how many bugs can you find in the below **`sList`**/**`slNode`** methods?

```
function slNode(val) {
  this.val = value;
  this.next = null;
}
function sList() {
  this.head = null;
  this.back = function() {
    if (!this.head) return null;
    var runner = this.head;
    while (runner)
    { runner = runner.next; }
    return runner.val;
  }
  this.pushBack = function(value) {
    var newNode = slNode(value);
    if (!this.head)
    { this.head = newNode; }
    else {
      var runner = this.head;
      while (runner.next)
      { runner = runner.next; }
      runner.next = newNode;
    }
  }
  this.popBack = function() {
    if (!this.head) { return null; }
    var returnVal;
    if (!this.head.next) {
      returnVal = this.head.val;
      this.head.val = null;
      return returnVal;
    }
    var runner = this.head;
    while (runner.next)
    { runner = runner.next; }
    returnVal = runner.val;
    runner.next = null;
    return returnVal;
```

```
}
```

```
this.pushFront = function(value) {
  var oldHead = this.head;
  this.head.next = oldHead;
  this.head = new slNode(value);
}
this.popFront = function() {
  var returnVal = this.head.val;
  this.head = this.head.next;
  return returnVal;
}
this.contains = function(value) {
  var runner = this.head;
  while (runner) {
    if (runner.val == value)
    { return true; }
    runner = runner.next;
  }
  return false;
}
this.removeVal = function(value) {
  if (!this.head)
  { return false;}

  if (this.head.val === value)
    this.head = this.head.next;
    return true;

  var runner = this.head;
  while (runner.next) {
    if (runner.next.val!==value) {
     runner.next=runner.next.next;
     return true;
    }
    runner = runner.next;
  }
  return false;
}
```

Chapter 7 – Linked Lists, Part II

Doubly Linked List

There is certainly no reason why a linked list node must refer to only one other node. For the best flexibility when traversing a list, we would want to be connected in both directions: forward and backward. Whereas singly linked lists feature nodes that only know about their forward neighbor (unable to look backward), *doubly linked lists* are more like lines of preschoolers holding hands as they walk down the street together, on a field trip to the fire station.

For the Doubly Linked List, all the concepts and techniques of Singly Linked Lists apply (see below). This extra flexibility comes with a cost, however. Maintaining both sets of pointers can be tedious.

classes and *objects* *private* vs. *public* *prototype* === vs. == *reference* vs. *value*

DList Class

Given the above reference implementations for doubly linked node and doubly linked list, can you construct the rest of a basic dList class? This would include dList methods push(),pop(), front(), back(), contains(), and size().

Second: implement these so that they are available as *standalone functions* as well as methods on both dlNode *and* dList classes.

```
function dlNode(value)
{
  if (!(this instanceof dlNode))
  {
    return new dlNode(value);
  }
  this.val = value;
  this.prev = null;
  this.next = null;
}
```

```
function dList()
{
  if (!(this instanceof dList))
  {
    return new dList();
  }
  this.head = null;
  this.tail = null;
}
```

Ninjas: as shown on the next page, implement functions we previously built for singly linked lists: **`isValid()`**, **`prependValue()`**, **`kthToLastElement()`**, **`deleteMiddleNode()`**, **`reverse()`**, **`isPalindrome()`** and **`partition()`**….

Chapter 7 – Linked Lists, Part II

Take on as many **`dList`** challenges as possible, before next chapter starts!

Prepend Value

Given dList, new value, and existing value, insert new val into dList immediately *before* existing val.

Kth To Last Value

Given k, return the value ‘k’ from a dList’s end.

Is Valid dList

Determine whether given dList is well-formed and valid: whether next and prev pointers match, etc.

Palindrome

Determine whether a dList is a palindrome

Loop Start

Given a dList that may contain a loop, return a pointer to the node where the loop begins (or null if no loop).

Repair

Combine previous work with a function that fixes errors found by **`isValid`** and breaks loops.

Append Value

Given dList, new value, and existing value, insert new val into dList immediately *after* existing val.

Delete Middle Node

Given a node in the middle of a dList, remove it.

Reverse

Create function to reverse nodes in a dList.

Partition

Given dList and partition value, perform a simple partition (no need to return the pivot index).

Break Loop

Given dList that may contain a loop, break the loop while retaining original node order.

Chapter 8 – Recursion

This chapter covers *recursion* and *dynamic programming*. Recursion occurs when a *function calls itself*. Dynamic programming is breaking large problems into smaller, more solvable ones.

Writing great code to solve a well-understood problem is only part of a software engineer's job. You should also consider how your code responds when given unexpected inputs. Thinking of possible "corner cases" ahead of time allows you to create much more resilient code that stands the test of time.

Why is *dynamic programming* useful? This is used when a function can make progress toward solving a problem, but is not able to solve the entire problem immediately. In this case, if we always make at least a little *forward progress*, then ultimately if we keep going, we will complete the computing task.

Let's consider an example: "I'm thinking of an integer between 1 and 120. Guess it." If you were to guess '60', and I said "nope, that's too high," how would you respond? You could treat the situation as if we were just starting, and I had said "I'm thinking of an integer between 1 and 59." In doing this, you reframed the problem as a less complex one, a technique called *dynamic programming*.

Let us continue: if next you guessed '30', and I said "nope, too low," then you could again reframe the problem as "I'm thinking of a number from 31 to 59." Let's say you then guessed '42', to which I said "yes, you guessed it, how did you know!" With this 'divide- and-conquer' approach, you could guess the correct number out of 120, in just 6 guesses on average.

There are three requirements for effective recursion, as follows:

a) **Base cases:**

When a function *can* determine (and return) an answer immediately, this is a *'base case'*. If you successfully guessed my number, we know right away the game is over. Conversely, if you look for *'spizzwink'* in a dictionary and find no word between *'spitz'* and *'splash'*, you know *'spizzwink'* is not in that dictionary. There are *positive* base cases and *negative* base cases.

b) **Forward progress:**

If a function *cannot* solve a problem but can narrow the range of possibilities, this is *'forward progress'*. Learning that guess '60' is *too high*, you have made forward progress because you now know the solution is not in the '60-120' range. Generally, for recursion to be effective, you must make at least a little forward progress in all cases. If there is a case in which you can neither solve the problem nor break it down further, you cannot solve the problem in all cases.

c) **Calling back into itself as if it were the original problem:**

What if earlier my initial challenge had been "I'm thinking of an integer between 1 and 59 -- guess it!" You would have proceeded exactly as you did in the original '1-120' problem, after learning that '60' was too high. If each guess were a function call, then after learning that '1-120' could be limited to '1-59', you could call the function again with '1-59' as if it were the original

challenge. Furthermore, this function wouldn't know whether this request was the initial challenge, or a subsequent call to itself after narrowing things down! A recursive function behaves correctly either way, so it doesn't know or care about this distinction.

Chapter 8 – Recursion

This chapter you will familiarize yourself with recursion. Some or all of the following important concepts will be used in this chapter's challenges.

base cases *forward progress* *call stack* *dynamic programming*

rSigma

Write a recursive function that, given a number, returns the sum of integers from one up to that number. For example, rSigma(5) = 1+2+3+4+5 = 15; rSigma(2.5) = 1+2 = 3; rSigma(-1) = 0.

rBinarySearch

Write a recursive function that, given a sorted array and a value, determines whether the value is found within the array. For example, rBinarySearch([1,3,5,6], 4) = false; rBinarySearch([4,5,6,8,12], 5) = true.

Chapter 8 – Recursion

This chapter you will familiarize yourself with recursion. Some or all of the following important concepts will be used in this chapter's challenges.

base cases *forward progress* *call stack* *memoization* *dynamic programming*

Recursive Fibonacci

Write **`rFib(num)`**. Recursively compute and return the *num*th Fibonacci value. As earlier, treat the first two (num = 0, num = 1) Fibonacci values as 0 and 1. Thus, **`rFib(2)`** = 1 (0+1); **`rFib(3)`** = 2 (1+1); **`rFib(4)`** = 3 (1+2); **`rFib(5)`** = 5 (2+3). Also, **`rFib(3.65)`** = **`rFib(3)`** = 2. Finally, **`rFib(-2)`** = **`rFib(0)`** = 0.

Binary String Expansion

You will be given a string containing characters '0', '1', and '?'. For every '?', either '0' or '1' characters can be substituted. Write a recursive function that returns an array of all valid strings that have '?' characters expanded into '0' or '1'. Ex.: **`binStrExpand("1?0?")`** should return **`["1000","1001","1100","1101"]`**. For this challenge, you can use string functions such as **`slice()`**, etc., but be frugal with their use, as they are expensive.

Chapter 8 – Recursion

This chapter you will familiarize yourself with recursion. Some or all of the following important concepts will be used in this chapter's challenges.

base cases *forward progress* *call stack* *memoization* *dynamic programming*

Recursive "Tribonacci"

Write a function **`rTrib(num)`** that mimics the Fibonacci sequence, but adds the previous *three* values instead of the previous *two* values. Consider the first three (num = 0, num = 1, num = 2) Tribonacci numbers to be 0, 0 and 1. Thus, **`rTrib(3)`** = 1 (0+0+1); **`rTrib(4)`** = 2 (0+1+1); **`rTrib(5)`** = 4 (1+1+2); **`rTrib(6)`** = 7 (1+2+4). Handle negatives and non-integers appropriately and inexpensively.

String In-Order Subsets

Create **`strSubsets(str)`**. Return array with all possible *in-order character subsets* of str. The result array itself need not be in a specific order. Given **`"abc"`**, return **`["", "c", "b", "bc", "a", "ac", "ab", "abc"]`**.

Chapter 8 – Recursion

This chapter you will familiarize yourself with recursion. Some or all of the following important concepts will be used in this chapter's challenges.

base cases *forward progress* *call stack* *memoization* *dynamic programming*

rFactorial

Given a number, return the product of integers from 1 upward to that number. If less than zero, treat as zero. If not an integer, treat as an integer. Mathematicians tell us that 0! is equal to 1, so make `rFact(0)` = 1. Examples: `rFact(3)` = 6 (1*2*3). Also, `rFact(6.5)` = 720 (1*2*3*4*5*6).

stringAnagrams

Given a string, return an array where each element is a string representing a different anagram (a different sequence of the letters in that string). Example: if given `"tar"`, return `["art", "atr", "rat", "rta", "tar", "tra"]`. For this challenge, you can use built-in string functions such as `split()`.

Telephone Words

On older telephones, the keypad associates letters with each numeral. Given a seven-digit telephone number, return an array of all possible strings that equate to that phone number. For reference, here is the mapping: `[2:ABC; 3:DEF; 4:GHI; 5:JKL; 6:MNO; 7:PQRS; 8:TUV; 9:WXYZ]` – for completeness, map 1 to `I` and zero to `O`. As an example, given the phone number 6937130 you should return an array of 1296 different strings, including “mydrifo” and “oyesido”.

Chapter 8 – Recursion

This chapter you will familiarize yourself with recursion. Some or all of the following important concepts will be used in this chapter's challenges.

base cases *forward progress* *call stack* *memoization* *dynamic programming*

rListLength

Given the first node of a singly linked list, create a recursive function that returns the number of nodes in that list. You can assume the list contains no loops, and that it is short enough that you will not 'blow your stack'.

All Valid N Pairs of Parens

Given the number of pairs of parentheses, return an array of strings, where each string represents a different valid way to order those parentheses. Example: given `2`, return `["()()", "(())"]`.

Chapter 8 – Recursion

These end-of-chapter challenges all assume that you have some familiarity with the game *chess*. If you don't, here is all you need to know for these challenges. Chessboards are square, with 8 rows of 8 squares each. Queens are one type of chess piece, and in a single move they can travel any number of squares in either of the horizontal directions (along a *row*), or either of the vertical directions (along a *file* or *column*), or either of the diagonal directions. A piece is considered under threat from a queen if it is situated in a square where a queen can directly move.

IsChessMoveSafe

Return if a chessboard square is threatened. **`isChessMoveSafe(intendedMove,queen)`** accepts an object indicating the location to check, and object of same type indicating location of an opposing queen.

Second-level challenge: accept an *array of queens*.

AllSafeChessSquares

Build on your solution to the previous challenge, to create **`allSafeChessSquares(queen)`** that returns all chessboard squares not threatened by a given queen.

Second-level challenge: accept an *array of queens*.

Eight Queens

Build on previous solutions to write **`eightQueens()`**. Return all arrangements of eight queens on an 8x8 chessboard, so that no queen threatens any other. What is the best way to return these results?

Second: write a helper function that displays the results returned, using console.log().

N Queens

Generalize **`eightQueens()`** into a function **`nQueens(n,xSize,ySize)`** returning all arrangements of N unthreatened queens on an X x Y rectangular board. That is, **`eightQueens() == nQueens(8,8,8)`**.

Chapter 8 – Recursion

(TBD) Where's the Bug? (recursion version)

Without peeking at previous code, how many bugs can you find in the recursive code below?

Chapter 9 – Strings, Part II

This chapter we revisit strings, having mastered the fine art of *recursion*. Remember that recursion is not an answer to every problem; everything that can be solved with recursion can also be solved without.

Recall that strings are *immutable*. Individual elements (characters) within the set cannot be altered, but an entire string object can be replaced by a new one. For this reason, one can (if it feels like cheating) execute a statement like `myString = myString + ", dude!";`. The entire myString object is replaced by a new one.

When string questions are asked in interviews, it is *always* worth asking whether you are allowed to use built-in string library functions. It might be the case that the interviewer just wants to know whether you are aware of the built-ins, and will readily allow you to avail yourself of them. In many other cases, though, the intention is for you to work in low-level primitives, often duplicating one or more built-in string library functions. These questions can be tedious, but they are still exceedingly common.

In challenges this chapter, do not use built-in string methods unless they are explicitly mentioned. This means you must work with strings simply as immutable arrays of characters. One exception: when capitalization is mentioned, you are allowed to use `.toLowerCase()` and `.toUpperCase()`.

String2WordArray

Given a string of words (with spaces, tabs and linefeeds), returns an array of words. Given `"Life is not a drill!"` return `["Life", "is" "not", "a", "drill!"]`.

Bonus: handle punctuation.

Longest Word

Create a function that, given a string of words, returns the longest word. Example: given `"Snap crackle pop makes the world go round!"`, return `"crackle"`.

Bonus: handle punctuation.

Reverse Word Order

Create a function that, given a string of words (with spaces), returns new string with words in reverse sequence. Given `"This is a test"`, return `"test a is This"`.

Bonus: handle punctuation and capitalization. Example: given `"Life is not a drill, go for it!"` you should return `"It for go, drill a not is life!"`

Unique Words

Given a string, retain *words that occur only once*. Given `"Sing! Sing a song; sing out loud; sing out strong."`, return `"Sing! Sing a song; loud; strong"`. Punctuation is part of the word: `"Sing!"` is not `"Sing"`.

Bonus: ignore punctuation and capitalization. Ex.: given `"Sing song! Sing a song; sing out loud and strong."`, return `"a loud and strong"`.

Chapter 9 – Strings, Part II

This chapter we revisit strings, now that we have mastered the fine art of *recursion*. Recursion is not an answer to every problem; although it can be quite valuable in certain situations.

Rotate String

Create a standalone function that accepts a string and an integer, and rotates the characters in the string to the right by that amount. Example: given `("Boris Godunov",5)`, return `"dunovBoris Go"`.

Is Rotation

Given two strings, return whether the second is a rotation of the first. Would you change your implementation if you knew that usually the two were entirely unrelated?

Censor

Create a function that, given a string and array of *'naughty words'*, returns a new string with all naughty words changed to "x" characters. Example: given **`("Snap crackle pop nincompoop!" , ["crack", "poop"])`**, return **`"Snap xxxxxle pop nincomxxxx!"`**.

Second: handle capitalization appropriately.

Bad Characters

Given two strings, the second string contains characters that must be removed from the first. Return the resultant string.

Chapter 9 – Strings, Part II

This chapter we revisit *strings*, having mastered the fine art of *recursion*. Remember: recursion is not an answer to every problem; everything that can be solved with recursion can also be solved without.

Is Permutation

Create a function that returns whether the second string is a permutation of the first. For example, given `("mister","stimer")`, return `true`. Given `("mister", "sister")`, return `false`.

Bonus: handle uppercase/lowercase.

Is Pangram

Return whether a string contains all letters in the English alphabet (upper or lower case). For `"How quickly daft jumping zebras vex!"`, return `true`. For `"abcdef ghijkl mno pqrs tuv wxy, not so fast!"`, return `false`.

All Permutations

Create a function that returns all permutations of a given string. Example: given `"team"`, return an array with the unique ***24*** strings including `"team", "meat", "tame", "mate", "aemt", "tmea", "etam", "atme"`, etc. How can you know that you covered them all?

Is Perfect Pangram

Create a function that returns whether a given string contains all letters in the English alphabet (upper or lower case) *once and only once*. Note: ignore punctuation and spaces. Given `"Playing jazz vibe chords quickly excites my wife."`, return `false`. Given `"Mr. Jock, TV quiz PhD, bags few lynx."`, return `true`.

Chapter 9 – Strings, Part II

This chapter we revisit strings, having mastered the fine art of *recursion*. Remember: recursion is not an answer to every problem; everything that can be solved with recursion can also be solved without.

Dedupe String

Remove duplicate characters (case-sensitive) including punctuation. Keep only the *last* instance of each character. Given `"Snaps! crackles! pops!"`, return `"Snrackle ops!"`.

Index of First Unique Letter

Return the index of the first unique (case-sensitive) character in a given string. Ex.: `"empathetic monarch meets primo stinker"` should return `35` (str[35] is 'k').

Unique Letters

Return only the unique characters from a given string. Specifically, omit *all* instances of a (case-sensitive) character if it appears more than once, including spaces and punctuation. Given `"Snap! Crackle! Poop!"`, return `"SnCrcklePp"`.

Chapter 9 – Strings, Part II

This chapter we revisit strings, having mastered the fine art of *recursion*. Remember: recursion is not an answer to every problem; everything that can be solved with recursion can also be solved without.

Num2String

Create a function that converts a number into a string containing those exact numerals. For example, given `1234`, return the string `"1234"`.

Bonus: include fractional values as well. Given `11.2051`, return `"11.2051"`.

Num2Text

Convert an integer into the English text for the number. Given `40213`, return `"forty thousand two hundred thirteen"`.

Bonus: include 4 fractional digits. Given `11.2051`, return `"eleven point two zero five one"`.

Chapter 9 – Strings, Part II

This chapter we revisit strings, having mastered the fine art of *recursion*. Remember: recursion is not an answer to every problem; everything that can be solved with recursion can also be solved without.

String Encode

You are given a string that may contain sequences of consecutive characters. Create a function to shorten a string by including the character, then the number of times it appears. For `"aaaabbcddd"`, return `"a4b2c1d3"`. If result is not shorter (such as `"bb"=>"b2"`), return the original string.

String Decode

Given an encoded string (see above), decode and return it. Given `"a3b2c1d3"`, return `"aaabbcddd"`.

Shortener

Given string and desired length, return a maximally readable string of exactly that length. Suggested sequence: 0) remove any leading or trailing spaces (or conversely, pad on both sides out to the desired length), 1) capitalize each word before removing spaces between words (starting from the back), 2) remove punctuation, starting from the back, 3) remove lower-case letters (vowels first), from the back, 4) remove upper-case letters, from the back.
Given `"It's a wonderful life! "`:
25: `" It's a wonderful life!  "`
20: `"It's AWonderfulLife!"`
15: `"ItsAWonderfulLf"`
10: `"ItsAWndrfL"`
5: `"ItAWL"`

Chapter 9 – Strings, Part II

(TBD) Where's the Bug? (strings version)

Without peeking at previous code, how many bugs can you find in the string-related code below?

Chapter 10 – Trees

This chapter we explore trees, and in particular **binary search tree** (BST), an important data structure. The BST is optimized for quickly finding/retrieving elements. A BST is similar to a linked list, in that it stores data elements within node objects. Let's compare a *doubly linked list node* to a *binary tree node*.

```
function dlNode(value) {
  this.val = value;
  this.prev = null;
  this.next = null;
}
```

```
function btNode(value) {
  this.val = value;
  this.left = null;
  this.right = null;
}
```

In a *doubly linked list*, each node has a value, plus pointers to two peers (*prev* and *next*). Similarly, in a **binary tree** each node has a **value**, plus pointers to two children, **left** and **right**. Just as with linked lists, these pointer attributes often reference another node, but can be null. Linked lists and binary trees always start with a single node; in a linked list we call it the head, in a binary tree we call it the root. The BST *structures the data* in a tree rather than a flat linear sequence.

A binary tree node can have a left child and/or a right child; each child might have left and/or right children of their own. An entire section of a family might descend from one sibling as opposed to another, similarly there are related subsets of a binary tree. These are (no surprise) called *subtrees*. We refer to all nodes stemming from the root node's left pointer as the root's left subtree, for example. By their basic definition, neither generic binary trees nor generic linked lists impose any specific order on where values must be located in them. There *is* a type of binary tree that adds structure, though

The **binary search tree** adds a requirement that for every node, all nodes in its left subtree must have smaller values. Similarly, its right subtree must contain only values that are greater than or equal to its value. This constraint holds for every node in the subtree, not just the direct children. These rules determine exactly where new children are placed in a BST. If "Grandparent" node<50> has a right child with value 75, then children of node<75> are appropriately constrained. Specifically, the entire left subtree of node<75> must have values between 50 and 75.

BST nodes without children are considered *leaf* nodes. Depending on its values, no node is required to have two children. Even in a tree containing many values, the root node might have a *left* or *right* pointer that is null (e.g. if the root contains the smallest or largest value in the tree, respectively).

Chapter 10 – Trees

This chapter's BST challenges start with the following reference definitions:

```
function btNode(value) {
  this.val = value;
  this.left = null;
  this.right = null;
}
```

```
function bst() {
  this.root = null;
}
```

Add

Create an add(val) method on the bst object to add a new value to the tree. This entails creating a btNode with this value and connecting it at the appropriate place in the tree. Note: BSTs can contain duplicate values.

Contains

Create a contains(val) bst method that returns whether the tree contains a given value. Take advantage of the BST's structure to make this a much more rapid operation than sList.contains() would be.

Min

Create a min() method on the bst class that returns the smallest value found in the BST.

Max

Create a max() bst method that returns the BST's largest value.

Size

Write a size() method that returns the number of nodes (values) contained in the tree.

Is Empty

Create an isEmpty() method that returns whether BST is empty (empty BSTs contains no values).

Chapter 10 – Trees

Self-Instantiating Classes

Implement today's challenges into the BST class below. Notice anything new? Why would we add this?

```
function btNode(value) {
  if (!(this instanceof btNode))
  { return new btNode(value); }

  this.val = value;
  this.left = null;
  this.right = null;
}
```

```
function bst() {
  if (!(this instanceof bst))
  { return new bst(); }

  this.root = null;
}
```

Answer: the additions to our **`btNode`** and **`bst`** classes are **`instanceof`** checks which make it so that these objects can be created with the following code:

```
var myBST = bst();          // no new needed
var aNode = btNode(42);     // no new
```

Today, add these additional methods to our bst class implementation:

Remove

Remove a given val. Return false if not found.

RemoveAll

Clear all values from the tree.

Is Valid

Construct an isValid() bst method to determine whether a tree has valid structure. Specifically, ensure nodes/values are appropriately located in left or right subtrees. This might be more tricky than it seems at first glance.

Add Without Dupes

Add a given value *only if it is not already found*. Return true if added, false otherwise. Remember Set Theory: this changes our BST from a multiset to a set. What other methods need changing, if we want our BST to be a true 'set' data structure?

Chapter 10 – Trees

You may have already realized that all Binary Search Trees are not equivalent. The arrangement of the nodes in the tree have a real effect on its efficiency. In referring to a tree's shape, we use the terms *depth* (also known as *height*) and *balance*, and a few chapters down the road we will work with trees of certain shapres that we refer to as *full* and *complete*. Let us explore these terms further.

Binary Tree Depth

A tree's **depth** is the length from root to farthest leaf including both. If we add nodes to a BST in random order, the tree grows in a relatively *balanced* manner – left and right subtrees will be about the same size, with mostly equal depth. If we add elements in *sorted* order, the tree becomes *unbalanced*, resembling a linked list in shape, and depth might approach the total number of elements. Even balanced trees often have a few "holes" where non-leaf nodes have one child that is NULL. However, the fewer the holes, the more **full** a BST is. Using the tree metaphor, a full BST is a very "bushy" tree, rather than thin and spindly. More nodes, contained in the same number of layers, makes for a better tree. Stated differently, given two binary trees with the same number of nodes, the one with less depth is always more efficient. Why is this?

Our answer lies in the reason that finding values in BSTs is so much quicker than finding them in SLists. Every node in a binary tree has branches, so there isn't a single path to follow from beginning node to end node. Furthermore, because BSTs are ordered, we always choose the correct direction at every fork. Instead of searching all values, the longest search is only the depth of the tree. This is why shallow, full trees are best. As data structures go, BSTs *rock* for fast retrieval.

Today, add these additional methods to our bst class implementation:

Height

Build a height() method on the bst object that returns the total height of the tree -- the longest sequence of nodes from root node to leaf node.

Is Balanced

Write an isBalanced() bst method that indicates whether our tree is balanced. A binary tree is balanced when *all* its nodes are balanced. A btNode is balanced if the heights of its left subtree and right subtree differ by at most one.

Chapter 10 – Trees

Binary Search Tree Traversal

Binary search trees are mostly used for their ability to fast-find, but sometimes we need to enumerate all its values. Here are three examples that illustrate why we might want different *orders* for listing the tree node values. 1) If we need to temporarily convert a BST to a different data structure, we may want it to list its values in numerical order. 2) If we want to duplicate a tree, we need to build it from the parent nodes downward, so we want to list parent nodes before child nodes. 3) If tree nodes might not represent values, but, say, components in a manufacturing process, where a component can only be assembled once its 'child' subcomponents are built, then we might want to list both child nodes before their parent node. These three examples are called *in-order*, *pre-order*, and *post-order* traversals, respectively. The terms pre-order and post-order refer to when a node is enumerated, compared to its children – if it is enumerated first, then this is *pre-order*; if it is enumerated last, then this is *post-order*. As always, enumerating a left child first, then the node itself, then the right child, is *in-order*.

Bst2Array

Create **`bst2Arr(bst)`** that outputs an array containing a BST's values, traversed in-order.

Second: create both **`bst2ArrPre`** and **`bst2ArrPost`** for those traversal methods.

Third: refactor to minimize code.

Lowest Common Ancestor

Given a **`bst`** and two values, return the value that is closest in the ancestry chain of both nodes.

Bst2List

Create **`bst2List(bst)`** that outputs a singly linked list containing the BST values in-order.

Second: create both **`bst2ListPre`** and **`bst2ListPost`** to traverse in those ways.

Third: refactor to minimize code.

Traverse Pre-Order Without Recursion

Given a **`bst`** object, console.log its values in pre-order, *without using recursion*.

Chapter 10 – Trees

Add these methods that use in-order traversal. You may or may not need an attribute **.parent**. If you do add this attribute, consider how you would need to change the other BST methods you've built to date.

Val Before

Create a bst method that, given a value that may or may not be in the tree, returns the value that is most immediately smaller. Examples: for tree {2, 5, 8}, valBefore(3) returns 2, and valBefore(8) returns 5.

Val After

Write a method on the bst class that returns the value immediately following the given one, even if that given value is not contained in the tree. Examples: for tree {2, 5, 8}, valAfter(3) returns 5; valAfter(8) returns null.

Node Before

Create a btNode method that, given a node that is in the BST, returns a *pointer to the node* with the most immediately smaller value. Examples: for tree {2, 5, 8}, nodeBefore(node5) returns the node containing 2, and nodeBefore(node8) returns the node containing 5.

Node After

Parallel to nodeBefore, write a btNode method that returns the node immediately following the given node (which is guaranteed to be in the tree). Examples: for tree {2, 5, 8}, nodeAfter(node2) returns the node containing 5; nodeAfter(node8) returns null.

Chapter 10 – Trees

Making BST a Fully Navigable Data Structure

In order to move from one node to its successor, sometimes we will need to traverse to a node's parent. How costly would it be to add a .parent pointer to our btNode class? Would we need to make any changes to the BST class itself?

The BST methods add(), isValid(), remove() and addNoDupes(), as well as the btNode method isValid(), plus both constructors would need to change to incorporate .parent into btNode. For example, the btNode constructor would need to include **`this.parent = null;`**. The btNode.isValid method would need to ensure that **`if (this.left)`**, then **`(this.left.parent == this)`**. The bst.isValid method should ensure that **`this.root.parent`** is always null. Even more complex, though, are the required changes to the add and remove methods, where we need to account for both sides when making a change to the BST (not unlike when making changes to a Doubly Linked List!).

BST With Parent

Create a **`btNode2`** data structure (and the necessary changes for an accompanying **`BST2`**) that adds a **`.parent`**. When is it more optimal? Is it worth the trouble? Work out the changes to prove it to yourself!

Chapter 10 – Trees

(TBD) Where's the Bug? (tree version)

Without peeking at previous code, how many bugs can you find in the tree-related code below?

Chapter 11 – Sorts

Why do we study the specific topic of *sorting*? Because sorting is an area where algorithm choices have significant and obvious ramifications for how well that code performs. Learning about sorting algorithms will equip you with techniques that you can use to analyze any set of code.

How do we judge *code quality*? There are many ways one might assess software. For most, a first thought is simply: *in good software, the features work*. That's reasonable, of course, but many aspects go into *whether something works as expected.* In addition to basic functionality, here are a few others. Is it resilient to unexpected inputs or malicious intent? Does it run in diverse environments, such as device form factors and/or different browsers? Is it trustworthy with users' confidential data? Are string messages handled appropriately so that the code can easily be localized into worldwide languages? Can it be easily configured and serviced by those maintaining and updating it post-release? Is it accessible for customers with visual/auditory disabilities? Is the source code understandable (not just by one, but also broadly across the team)? How well documented is the source (or the overall product)? How rapidly/easily can it be extended for new features? Product excellence has many dimensions.

In many situations, what trumps most of these factors is software *performance*. This can also mean many things, but primary measures of software performance are *run time* and *resource consumption* (*time and space*, if you will). Everyone that works with software has experienced software that is frustratingly slow. Even small time optimizations can make a product feel more snappy and responsive. Regarding resource consumption, this can be permanent storage (storage, database), memory (heap, call stack), network bandwidth, or even power – after all, no one likes an app that drains their battery. Depending on product requirements, any or all of these may be important. That said, classic software analysis focuses on 1) run time and 2) memory consumption, when it comes to evaluating algorithms.

When comparing run times and memory consumptions of two piece of software, we must be careful. What if one is in PHP, and the other is in C or even machine assembly language? What if we run one on an Apple Watch, and the other on a massive IBM 64-processor server with 256 GB of RAM? To bend the "apples and oranges" metaphor, this would literally be comparing Apples and (Big) Blues. To factor out these differences and focus only on the algorithm, analysis is always relative, not absolute. Specifically, we compare the algorithm, in that language within that environment, to *itself*, when given different types and sizes of inputs. Specifically, by what percentage does run time change when we double the input size? By what factor does memory consumption grow, if we make our input ten times bigger? As input sizes get larger, extraneous factors melt away, leaving only critical ones that ultimately constrain whether our software can handle 100 simultaneous users, or 1,000, or even 100,000.

Certain algorithm texts refer to "asymptotic behavior" of an algorithm. This simply means the behavior of an algorithm as the (input) data set gets extremely very large. Almost any sorting algorithm will suffice if we are sorting only ten elements, but if we must sort 4 billion numbers, then algorithm choice matters. Later this chapter, you will learn more about how we measure algorithms and what leads to high-performance. Sorting algorithms are not the only software whose performance should be

analyzed, nor generally the most important. However, studying them is an excellent proxy for other software elsewhere. Have fun this chapter!

Chapter 11 – Sorts

We start with the first sorting algorithms programming students learn – Insertion and Selection.

Bubble Sort Array

For review, create a function that uses BubbleSort to sort an unsorted array in-place.

Selection Sort Array

For review, create a function that uses SelectionSort to sort an unsorted array in-place.

Bubble Sort List

Create a function that uses BubbleSort to sort a singly linked list. The list nodes contain **`.val`**, **`.next`** and other attributes that you should not reference.

Selection Sort List

Create a function that sorts a singly linked list using selection sort. List nodes contain **`.val`**, **`.next`** and other attributes that you should not reference.

(TBD) Multikey Sort

Chapter 11 – Sorts

Big-O Notation

Previously we mentioned that when analyzing algorithms, comparisons must be relative, when given different inputs. Specifically, as we increase the input size by 10, how does the time needed to run the algorithm change? How does memory consumption change? Generally, there are only a few growth rate types. The mathematical convention that represents these growth factors is called Big-O notation.

Side note: really hard-core algorithm analysis experts talk not only about Big-O but also *Big-Omega* and *Big-Theta*. In brief, Big-O describes *worse-case* performance; Big-Omega *best-case;* Big-Theta *average case*. Their values can differ, but for our purposes you can think of them as the same. Further, when best-case and worst-case differ, most people talk about Big-O and then mention "best-case".

What does Big-O notation indicate? It conveys how an algorithm will perform, as input sizes grow large. As we multiply our input by factor N, how do our run time and memory consumption change? In practice, we would first measure the time and memory consumed when running with an input of specific *size*; then measure the same when given an input of "size x N" -- this ratio, in terms of N, is our Big-O.

Consider **`FindMax()`** that returns an array's lowest value. If we double the array length, we expect **`FindMax`** to run twice as long. If we multiply array length by N, run time should multiply by exactly N as well. Hence we say the time complexity of this algorithm is O(n), or verbally "for run-time, it has a Big-O of N." Looking at memory consumption, the only memory needed is local storage of a FOR loop index, and a local variable to track the min value. This is the case *regardless* of the array's length, so as we multiply array length by N, our additional memory requirements are constant (i.e., multiplied by **1**). Hence the algorithm's memory complexity is O(1), or verbally "for memory, Big-O is 1." If our algorithm needed to make a copy of the array, then the Big-O for memory would be O(n). One last thing: with recursion, we also factor in the additional stack space as we recurse. We'll briefly touch on that later.

Insertion Sort

Create a function that InsertionSort to sort an unsorted array in-place. What is the run-time complexity? What is the space complexity?

Partition Array

Partition an unsorted array in-place. Use first element as pivot; return index where pivot ended. For [5, 1, 8, 4, 9, 2, 5, 3], change array to [1, 4, 2, 3, 5, 8, 9, 5] and return 4.

Second: For pivot, use median of *first*, *last*, *mid*.
Third: Partition a *subset*, given *start* and *end*. Exclude end; default values are 0 and arr.length.

Insertion Sort List

Use InsertionSort to sort singly linked lists. Only reference list node attributes .val and .next. What are the run-time and space complexities?

Partition List

Partition a singly linked list. Use first element as pivot; return the new list. List nodes contain .val and .next; do not reference other attributes. For example, given { 5=>1=>8=>4=>9=>2=>5=>3 }, return the list { 1=>4=>2=>3=>5=>8=>9=>5 }.

Chapter 11 – Sorts

When discussing sorting algorithms, we talk mostly about *run-time* performance – *how does Time Needed change, as input size grows?* But Big-O can also refer to resource consumption: memory, storage or network bandwidth – most commonly, RAM. The memory consumed by an algorithm is either heap or call stack or both. These correspond to 1) copying an input or 2) making recursive calls. Clearly, all things equal, an algorithm shouldn't make a copy of the input. After all, we might receive an array containing 4 billion values! Also, call-stacks are not unlimited; it doesn't take much to "blow our stack". Everything solved with recursion is solvable without. More on O(space) later.

So which sorting algorithm is *truly* best? Again the only correct answer is "depends on situation." There are a number of characteristics, though, that describe sorting algorithms, and we will discuss one of these each day through the rest of the chapter. Today we discuss what makes an algorithm *Adaptive*.

Adaptivity

For many algorithms, the input data's configuration (e.g. randomized, mostly sorted, reversed) makes a big difference in performance. Even when handed already-perfectly-sorted data, SelectionSort makes the same number of comparisons as if values were in random order. There is almost zero difference between its best-case performance and worst-case performance. (Good news: predictable! Bad news: O(n2)!) So, SelectionSort does not *adapt* to data that is already partially or fully sorted.

On the other hand, InsertionSort and BubbleSort show huge differences between best-case and worst-case run time. You can even see this in the code, whenever we have a "fast finish" check that breaks out early if we make a complete pass without needing any swaps. We call these algorithms Adaptive.

When is this important? Consider a huge quantity of existing, already-sorted data, where you must add a small number of new, unsorted values. With InsertionSort, we can quickly sort these new values into place with little penalty from the magnitude of existing data. Big win!

QuickSort (array)

Create a function that uses yesterday's PartitionArray to sort an array in-place. With yesterday's code plus very few new lines, you've implemented QuickSort! What are the run-time and space complexities of QuickSortArr?

Combine Sorted Arrays

Combine two already-sorted arrays, returning a new sorted array with all elements.

Partition3

Previous Partition implementations do not group pivot dupes together. Create `partition3()` to keep duplicate pivot elements together; return a two-element array containing indices for first pivot and first greater. Change `[5,1,8,4,9,2,5,3]` to `[1,4,2,3,5,5,8,9]` and return `[4,6]`. Note: other 5 moved next to pivot.

Second: Pick a more optimal pivot.

Third: Partition only a *portion*, with *start* and *end*.

Chapter 11 – Sorts

We've discussed what makes a sorting algorithm *Adapt* Can it sometimes run faster because it takes advantage of inputs that (for whatever reason) are already at least partially sorted? Today we discuss a new characteristic: stability. What is it, and why does it matter?

Stability

In most of our sorting work so far, we have dealt with simple collections of single values, such as an array of 15 numbers. In real life, however, data is rarely that minimal, and never that simple. Much more likely would be a linked list or array containing many thousand records, and each record contains 5-10 different fields. To sort these records, we may need to reference multiple fields (for example: sorting customer events by last name, then first name, then event date). One strategy, when sorting by multiple fields, is to sort first by the least important factor (date), then by more important factors (first name), ending with the most important (last name).

However, this multipass strategy *only* works if our sorting algorithm is able to retain the existing order, when finding duplicate values during subsequent passes. Some algorithms, such as SelectionSort and QuickSort, swap elements across significant parts of the input array, and hence do not guarantee to keep duplicate values in their original sequence; they *destabilize* any preexisting ordering. (If we have already sorted "Alan Jones" ahead of "David Jones" based on first name, we don't later want to carelessly put David Jones ahead of Alan Jones just because their last names are identical. With Selection and Quick, this might happen!) On the other hand, InsertionSort and Bubblesort only swap adjacent elements, so existing sequence is preserved when they encounter duplicate values. InsertionSort and BubbleSort are Stable sorting algorithms. Again, this becomes a factor mainly when doing successive sorting passes, such as sorting by multiple fields (e.g. userID, date).

Combine Sorted Lists

Create a function that combines two already-sorted linked lists, returning a sorted list with both inputs. List nodes contain .val and .next, as well as other attributes that you should not reference.

Merge Sort List

Use CombineLists from yesterday to construct the MergeSortL algorithm for an unsorted singly linked list. What are the run-time and space complexities of your MergeSortList solution?

Merge Sort Array

Use yesterday's CombineArrs function above to construct mergeSortA() for an unsorted array. What are the run-time and space complexities of your MergeSortArr solution?

(TBD) Making a Sort Stable

Chapter 11 – Sorts

As mentioned before, when we analyze the performance of a piece of software, we look at runtime performance (how long does it run to completion), but also how hungry for other resources. Does it require a huge amount of disk storage space? Is it excessively chatty, networking-wise? Will it mercilessly drain our battery in mere minutes? An important factor (and sometimes easiest to quantify) is the amount of extra *memory* (in addition to input data) it requires. Earlier we mentioned that RAM consumption could be *heap* space (e.g. we allocate enough space to hold a copy of our array!), or *call stack* space (we recursively call ourselves, once for every node in our list!). Either way, we might run out of memory, at which point (depending on the environment where we are running our algorithm) our code will slow to a crawl (best-case) or crash (worst-case, and common).

Memory Analysis

From a call stack perspective, we simply cannot use recursion unless we limit the depth of any single chain of recursion. For example, if we have a BST with 1,000,000 elements, nonetheless the tree (on average) will only be 20-25 elements deep, so recursively traversing a BST consumes a tolerable amount of stack space. Specifically, we would say that it requires O(logN) stack space. (Why? Think of log as the inverse of exponents. Each time our tree goes one level deeper, it doubles in size. How many levels do we need to handle one million? Well, 2 raised to the 20 power is ~ 1 million. In other words the log2 of 1 million is 20, so on average a BST of size N would be logN nodes deep.)

What about heap space, then? What if we don't make a copy of the data, but we do maintain a bunch of local variables, maybe a dozen? Not a problem! In the big picture, don't worry about a dozen local variables -- their 100 puny bytes are insigificant compared to the power of the force the magnitude of even just copying half of the potentially-huge input data we are sent. The ideal algorithm, then, is one that requires no additional copy of the input data at all. If an algorithm can operate successfully with no additional heap requirements, but only the space of the input data it is handed, then we call that algorithm in-place. This does not mean that the algorithm is automatically O(1) as far as memory is concerned, because if it is recursive it might consume stack space. However, running in-place is a huge win in many scenarios, particularly on mobile devices with limited memory.

Counting Sort

Believe it or not, you're given have an array containing the IQ of every person on earth! Create a function that takes this unsorted collection and sorts it in-place. All array values are between 0 and 220. Remember: the array is 7+ billion elements long.... What are the run-time and space complexities of your CountingSort implementation?

QuickSort3

Create a QuickSort3 function that uses Partition3 to sort an array in-place. Can you devise specific arrays that are sorted much faster with QuickSort3 than with QuickSort?

(TBD) Master Directory from Departments

Chapter 11 – Sorts

Sorting Review

We discussed numerous aspects that different parties might consider important in judging a piece of software “good”. These include correctness, resiliency to bad inputs, security against hackers, being easy and fun to use, extensibility to accomodate future features, clarity of being understood by other engineers, whether the software is easily internationalized to 100+ spoken languages around the world, how easily it can be deployed or updated, etc. Depending on your role, “good” software means different things. For most, though, performance is in your top 3 most important factors.

We focus on performance of *sorting algorithms* because there are multiple diverse ways to sort data, and most algorithms have at least one redeeming factor making it valuable in some situation. We need to know when to choose that particular algorithm, based on the problem’s requirements.

For algorithm performance, we factor out HW specifics and programming language, only comparing and only compare an algorithm to itself, with some other larger input data. The headstart from having a faster programming language is insigificant compared to the power of a superior algorithm (if our data set is big enough). As our input data grow larger by N, by what factor does runtime grow? By what factor does memory requirement grow? This type of analysis is called Big-O notation.

We focused primarily on the Big-O of an algorithm’s *runtime* (how long it takes to run), but also memory usage. Runtime complexity is often understood by looking at nesting of loops. O(n2) algorithms such as BubbleSort are considered slow. The basic algorithms used to teach sorting are all O(n2) because, essentially, they compare every value to every other value (N x almost-N). However, some algorithms have optimizations that lead to significant differences between average run-time and best-case runtime. More sophisticated algorithms use “divide and conquer” schemes that quickly cut the problem space so that each value is not compared against every other. In this way they roundly defeat O(n2) algorithms -- their average performance is O(nlogn). That said, they might have weaknesses, such as specific input data that trigger horrible worst-case performance (QuickSort).

We discussed specific characteristics of sorting algorithms, and when they might be important. These include being ***adaptive*** (taking advantage of partially sorted data), ***stable*** (retaining existing sequence of duplicates), ***in-place*** (not using space beyond the input data). None of the O(nlogn) algorithms are adaptive, and although MergeSort is the only one that is stable, it is also the only one *not* in-place.

Memory usage might be in the *heap* (from copying the input data), or the *stack* (from recursive “divide and conquer” calls). On mobile devices memory is often scarce; operating in-place is usually critical.

In special situations you can use unusual sorting algorithms such as CountingSort, which miraculously sorts simple values in O(n), but cannot accommodate additional data. Understanding the various characteristics of these algorithms is what enables you to choose the right tool for the right job.

RadixSort

For an array 7 million long with values from 0 to 4 billion, how rapidly can you RadixSort it in-place? You can create a new array as large as original. What are run-time and space complexities?

Median of unsorted array

Create a function that determines the median of an array of unsorted values. Doing this in O(n) is agreed-upon as a ‘hard problem’.

(TBD) Urban Dictionary Daily Add

Pancake Sort

Your Dojo classmate cooked you breakfast, and you have a large stack of pancakes of varying widths. Your goal is *obviously* to stack them from widest (on bottom) to thinnest (on top), because syrup pours best that way. Your only tool is a spatula that you can insert below any pancake and flip all pancakes on top. The pancake widths are represented in an array: for example, four pancakes that are already stacked from biggest to smallest would be `[4,3,2,1]`. If you insert the spatula between second and third pancakes and flipped, that stack would now be `[4,3,1,2]`. Given an arbitrarily large stack of N pancakes, how many spatula flips will it take to sort the pancakes into width-order? Design a high-performance algorithm, because everyone is getting hungry….

Chapter 11 – Sorts

(TBD) Where's the Bug? (sorts version)

Without peeking at previous code, how many bugs can you find in the sort-related code below?

Chapter 12 – Sets and Priority Queues

Sets and Multisets

Whether working with a deck of cards or results from a database query, we constantly work with *sets* - a mathematical term for collections of values that we group together. Just as there are many reasons to group values together, likewise there are different types of sets, each useful in certain situations. Specifically, you might care how a set handles *duplicates*, and whether it keeps values *ordered*.

By default, ***sets*** do not contain duplicate values; adding value 42 to the set ("Zork", "grue", 42), you still have ("Zork", "grue", 42). Ex.: when gathering nominations for Best Restaurant in Town, the nominee list is a *set*. There can also exist sets that contain duplicate values; these are ***multisets***. In collections of this type, duplicate values matter: multiset (1, 1, 1, 3) and multiset (1, 1, 3, 3) are *not* equivalent. Example: after a public vote for favorite restaurant, during the vote-counting process we could use a multiset, such as (Joe's, Joe's, Mel's, Joe's, Joe's...).

For some sets, the order matters (such as words in a dictionary). This type is called an ***ordered set***. For others (e.g.: members of a club), it doesn't: these are ***unordered sets***. More on these tomorrow!

What can we do with these different flavors of set? You might *add* an element, *remove* one, or check whether the set *contains* a certain element, and if so how many. With *ordered* sets, we can also retrieve *first* or *last* element, and from any element get *next* or *previous*. These are standard operations for any data structure that we use to collect values, such as an array or a singly linked list.

Throughout the chapter, remember these set types:

set *multiset* *ordered* *unordered*

Interleave Arrays

Given two unsorted arrays, create a new array containing the elements of both, resulting in an *unsorted merge* of all values. When populating the new array alternate (interleave) values between the two arrays until one is exhausted, then include all of the other. Example: given `[77,22,11,22]` and `[2,6,7,2,6,2]`, return `[77,2,22,6,11,7,22,2,6,2]`.

Merge Sorted Arrays

Efficiently merge two already-sorted arrays into a new sorted array containing the *multiset* of all values. Example: given `[1,2,2,2,7]` and `[2,2,6,6,7]`, return `[1,2,2,2,2,2,6,6,7,7]`.

Chapter 12 – Sets and Priority Queues

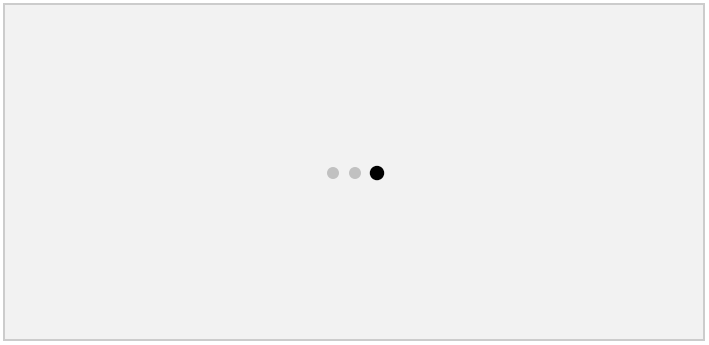
Set Operations

Yesterday we discussed different characteristics of a *set*. By default, sets contain no duplicates, but a type that can is called a *multiset*. A normal set does not track the counts of values, but a multiset does.

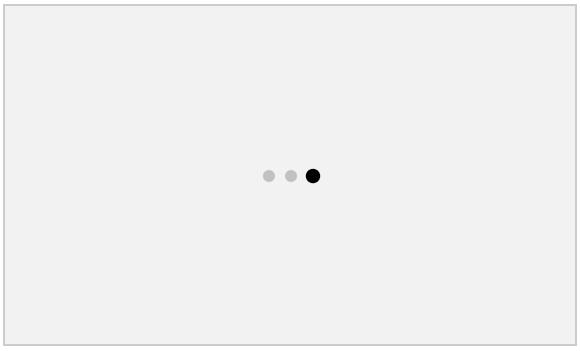
Sets that keep elements in strict order are *ordered sets* (or *ordered multisets!*). Those that don't are *unordered sets/multisets*. There are costs associated with removing duplicates or maintaining a set's order, so if we add values without sorting or removing duplicates, we have an unordered multiset.

When working with more than one set, we can combine them in various ways. We can simply ***merge*** two sets, resulting in a multiset that includes all values from both sides. Merging always results in a multiset, because every duplicate is kept. A second option is to get the ***union*** of sets A and B, which would include all values from one set, plus anything from the other set that we haven't already included. Conceptually, this equates to a logical OR: to be included in this union, an element must be found in one set *or* the other (or both). A third type of set combination is an ***intersection***, conceptually similar to a logical AND. To be included in an intersection, an element must be found in one set *and* the other.

A *merge* is simply the sets, together.



A *union* contains values in either set: `[1,3,1,2,6]`



An *intersection* contains only values in both sets: `[1]`

Shown above, for *multisets* these operations affect *value counts* accordingly. For multisets containing the same value ('1' above), union retains the higher count (3 '1's are retained); intersection retains the lower count (2 '1's are retained). Given ordered multisets `[1,1,1,3]` and `[1,1,2,6]`: the *merger* is

`[1,1,1,1,1,2,3,6]`; the *union* is `[1,1,1,2,3,6]`; the *intersection* is `[1,1]`. Lastly: if a set contains all the values of another set, like `[1,2,2,3]` and `[2,3]`, the second is a ***subset*** of the first.

Intersect Sorted Arrays

Efficiently combine two sorted arrays into an array containing the sorted *multiset intersection* of the two. Example: given `[1,2,2,2,7]` and `[2,2,6,6,7]`, return `[2,2,7]`.

Intersect Sorted Arrays (dedupe)

Efficiently combine two sorted multiset arrays into an array containing the sorted *set intersection* of the two. Example: given `[1,2,2,2,7]` and `[2,2,6,6,7]`, return `[2,7]`.

Chapter 12 – Sets and Priority Queues

As you proceed through this chapter, put yourself in a technical interview mindset with these concepts:

Don't panic think out loud clarifying questions error and corner cases example inputs diagrams admit when its suboptimal (but keep going) "what are we optimizing for?"

Throughout these challenges, remember the basic set operations and characteristics:

merger union intersection set / multiset ordered / unordered subset

Union Sorted Arrays

Efficiently combine two already-sorted arrays into a new sorted array containing the *multiset union*. Example: given `[1,2,2,2,7]` and `[2,2,6,6,7]`, return `[1,2,2,2,6,6,7]`.

Intersection Unsorted Arrays (in-place)

Intersect two unsorted arrays, putting the *unsorted multiset* result 'in-place' into the first. Running 'in-place' also means you cannot create any data structure to hold values, such as an associative array. Given `[2,7,2,1,2]` and `[6,7,2,7,6,2]`, you could change the first to `[7,2,2]` in any order.

Union Sorted Arrays (dedupe)

Combine two sorted arrays into a new sorted array containing the *union set* (i.e. remove duplicates). Example: given `[1,2,2,2,7]` and `[2,6,6,7]`, return `[1,2,6,7]`.

Intersection Unsorted Arrays

Intersect two arrays to create an *unsorted multiset*. You can use ad additional data structure type if it is helpful. However, don't alter the arrays; return a new one. Given `[6,7,2,7,6,2]` and `[2,7,2,1,2]`, return a new array containing `[7,2,2]` in any order. Is 'non-in-place' easier? Faster?

Union Unsorted Arrays

Return a new *unsorted union multiset* of two arrays; do not alter the originals. For `[2,7,2,1,2]` and `[6,7,2,7,6,2]`, you could return `[7,2,7,2,2,1,6,6]`. How efficient can you be, for *long* arrays?

Chapter 12 – Sets and Priority Queues

This chapter we dive further into Set Theory. Put yourself into the mindset of a technical interview during this chapter's algorithm challenges, using the following concepts:

merger *union* *intersection* *set / multiset*
ordered / unordered *in-place* *stable* *subset*

If needed, refer to your previous solution to "Union Unsorted Arrays" challenge for starting points:

Union Unsorted Arrays (in-place)

Put *union multiset* of 2 unsorted arrays into first. Given `([2,7,2,1], [6,7,2,6])`, change the first array to include (in any order) the elements `[2,7,2,1,6,6]`.

Union Unsorted Arrays (no duplicates)

Return the *union set* (remove any duplicates) of two unsorted arrays. Given `([2,7,2,1], [6,7,2,6])`, return (in any order) `[2,7,1,6]`.

Set Theory Recap

This chapter we explored Set Theory, with concepts such as Sets vs. Multisets and Ordered vs. Unordered, as well as set operations such as Merge, Union and Intersection. Along the way we discovered a few universal principles, such as:

- Ordered sets must be managed by iterating them concurrently, matching up values.
- Unordered sets can be managed with associative arrays, where set members become keys, and values are Boolean (for an Unordered Set) or counts (for an Unordered Multiset).

Next, we revisit our old friend the *queue*, but with a wonderful twist!

Chapter 12 – Sets and Priority Queues

Queues and Stacks are easy to construct, and values can be quickly added and removed. They are, in fact, *optimized* for quick addition – and for quick removal as well, *if you want values extracted in order they were added* (or *reverse* order, for Stacks). However, they are not optimized for quick search: elements are stored linearly based on insertion time, without regard for the values themselves. Hence, we might (for example) iterate every value before finding the lowest one.

Priority Queues

What if, instead, we created a data structure that acted like a Queue, but *did* care about values instead of insertion time. This data structure would maintain its elements in *value* order, regardless of the order in which they were added. We could extract (pop, dequeue) an element at a time, and always get the lowest value. This is be valuable as (for example) a way to prioritize a list of todo items so that when we take an item from the Queue, it is always the most important one. In fact, OS subsystems such as networking and storage work in this way: diverse I/O requests are continually added to a prioritized queue, and the system extracts (and satisfies) them in priority order. Let us build this *Priority Queue*.

Singly Linked List Priority Queue

We want to create a Queue data structure that keeps its elements in sorted order, so that when we call pop(), we get the first element in *sorted* order (rather than sequential order, like a regular FIFO queue).

Create a priQueue data structure by making the following changes to slQueue and slNode:
A priQNode class should be identical to slNode, plus a .pri attribute which is set by an additional argument passed to the priQNode constructor. The priQueue push() method should accept both value and *priority*, and priority should be used to add the node at the right spot (instead of at queue's end).

Sequencer

Using a singly linked list priority queue object, build a system that orders and "plays" messages uses the system timestamp (get this by calling `Date.now()`). Create two functions that are used as follows:

```
sequenceMessage([2000000000000, "Msg 4"]);
sequenceMessage([1453506544890, "Msg 2"]);
sequenceMessage([1453506544900, "Msg 3"]);
sequenceMessage([1000000000000, "Msg 1"]);
// assume current time is now 1453506544898
playMessages();       // "Msg 1", then "Msg 2" are logged to console
// assume current time is now 1453506544915
playMessages();       // "Msg 3" is logged to console
```

The **sequenceMessage(arr)** function will be sent a two-element array, containing a timestamp and a string. The timestamp is in milliseconds, and corresponds to values obtained by Date.now(). You should sort these messages by ascending timestamp. When **playMessages()** is called, console.log (in order) the strings of messages with timestamps in the past, and remove them from your list.

Heap Data Structure

A mythical beast called the *manticore* had a lion's body, a human head, and a scorpion's tail. A priority queue is commonly constructed with a *minheap*, which has similarly unusual characteristics. (This *heap* is not the heap where memory allocations occur, although they use the same word.) Not to be outdone by manticores, heaps behave like queues, manage data like BTrees, and are stored in Arrays. Rather than extraordinary speed in only a few aspects, heaps strike a balance: great insertion, good deletion, and great extraction (in monotonic priority order). So, how does this creature do it?

Interestingly, the heap isn't fully sorted. It is "just sorted enough" so that it can extract the lowest value immediately, and quickly rearrange other values tree so that it is again "just sorted enough" for the next extraction. Insertion, similarly, works "well enough" to keep the tree somewhat sorted, without doing extra work. This extraordinary laziness leads to high performance.

For these purposes, we will assume we are building a *minheap*, although the rules are easily inverted if we want a maxheap. The only difference in behavior is that maxheaps extract values from largest to smallest, instead of lowest first for a minheap. Otherwise they follow the same rules.

First, data in a heap are arranged in binary nodes. *Second*, in a minheap every node must have a value less than or equal to its children (greater than or equal, for a *max*heap). *Third*, the minheap forms a *complete* binary tree, where every node has two children except for the deepest level, where nodes are present starting from the leftmost extending toward the right. That's it for the rules.

Here's the next interesting detail: instead of using actual binary tree nodes, the minheap puts values into an array, and uses the array indices to track the parent-child relationships between values. Specifically, a value at index N has children at indices 2N and 2N+1, and its parent can always be found at N/2. The root of the heap is located at index 1 (index 0 usually holds some other value). In this way, tree traversal from an arbitrary node is quick. With this in mind, four our six basic data structure methods (size(), isEmpty(), top(), contains(val)) are trivial. The first three are immediate, but the performance of contains(val) is horrid (effectively, search the entire underlying array). The insert(val) and extract() methods are the most interesting.

For insert(val), we know that our tree's size will grow by one. Because it is a complete binary tree we know exactly where a new "node" will be added (we know our array.length will grow by one). We don't know what value should go there, but we start by putting the *new value* in that spot. We then go through a "promote" process for that value, where we compare it to its parent. If its value is less than that of its parent, we swap them and continuing trying to promote the new value (comparing to its new post-swap parent). Once it can no longer be promoted, the insertion process is complete.

Constructor

Create a minHeap constructor function.

Is Empty

Return whether the heap is empty.

Size

Return the number of values in the minHeap.

Top

Return (not remove) the heap's minimum value.

Contains

Return whether given val is within the heap.

Insert

Create a method that adds a value to our heap.

Chapter 12 – Sets and Priority Queues

Let us continue our development of the minHeap data structure. Previously, we developed the ability to add elements. Now we will build a method to remove the top element – we'll call it extract().

For this discussion, keep in mind that although we are storing the element values in an array, we are still thinking about the collection of values as a binary tree. With this in mind, below I refer to values and nodes. By nodes, I just mean the array index where the value is found.

For extract(), we know that our tree size will shrink by one node. Because it is always a *complete* tree, we know that the node to be removed is the *last* node. In other words, our array will become shorter, by one. We also know which *value* needs to be removed from our array, and it is the *first* value, not the last value. To use a metaphor, we remove the first person, and the last chair. So our challenge, when we extract a value from our heap, is how to *minimally* rearrange values in the tree so that all the remaining nodes are occupied by the remaining values, in a way that satisfies all the heap rules. Doing this in a minimal way is the hallmark of a heap.

We start by considering the last value in the heap – the one sitting in the array index we want to remove. We give that value an unusual opportunity: we move it to the *root* for a short time. From there, we will put the value through a "demote" process to shift it downward in the tree to a more suitable spot. What does this "demote" process entail? After swapping the value into the root spot, we first compare it to its two new children. If either of them has a lower value than it does, we swap it with the lower one, then repeat this "demote" process with that same value in its new spot, until it has no children with lower values (this might not happen happen until it has no children at all!). Once it can no longer be demoted, the extraction process is complete.

With Extract(), our basic data structure implementation is complete. Additionally, we would like the ability to pass in an array and have the Heap adopt that array as its own, quickly repairing it to a state of compliance with the Heap rules. First, change the Heap constructor to optionally accept an array. Also create the Heapify method that accepts and repairs this array after the Heap has been created.

Extract

Create a heap method that removes the heap's minimum value and returns it.

Heapify

Create a heap method that accepts an array as its own, and turns it into a rule-abiding minHeap.

Heap Sort

Miraculously, if one heapifies an unsorted array then extracts values, the array is sorted in O(NlogN) time - as fast as QuickSort & MergeSort, the winners in generalized sorting! This proves that the Heap truly is "the crown prince of data structures." Write a standalone function, unlike the methods above, to create **`heapSort(arr)`** – a function that accepts an unsorted array and uses a heap to sort it.

Second: do this in-place without creating a second array

Chapter 12 – Sets and Priority Queues

Before next chapter, here are a few other Queue/Stack problems to keep you thinking. Have fun!

Queue from Two Stacks

Using only stack objects (not other data structures such as linked lists or arrays), implement a queue.

Comparing Stacks/Queues to Other Data Structures

By now we have studied a few different data structures: array, singly and doubly linked lists, binary search tree, slQueue, cirQueue, array-based stack, deque, priority queue. Each of these could be built as a Set instead of a Multiset (rejecting duplicate values instead of accepting them). We will not require you to build all the possible variants, but below we list them for completeness. Those that are bolded are ones you've already built previously; those underlined are highly recommended. In most cases, creating these will require only small adjustments to code you've already written.

Array (random-access multiset)
Array without duplicates (random-access set)

SList (forward-iterated insertable multiset)
SList without duplicates (forward-iterated insertable set)
DList (double-iterated insertable multiset)
DList without duplicates (double-iterated insertable set)

Binary Search Tree (ordered multiset)
Binary Search Tree without duplicates (ordered set)

SLQueue (sequential multiset)
SLQueue without duplicates (sequential set)
CirQueue (sequential multiset)
CirQueue without duplicates (sequential set)

SLStack (sequential multiset)
SLStack without duplicates (sequential set)
ArrStack (sequential multiset)
ArrStack without duplicates (sequential set)

Deque (double-sequential multiset)
Deque without duplicates (double-sequential set)

PriQueue (forward-ordered multiset)
PriQueue without duplicates (forward-ordered set)

Next chapter we'll build these:
AssociativeArr (unordered multiset)
AssociativeArr without duplicates (unordered set)

Chapter 12 – Sets and Priority Queues

(TBD) Where's the Bug? (set/heap version)

Without peeking at previous code, how many bugs can you find in the set/heap-related code below?

Chapter 13 – Hashes

Have you ever wondered how a *key-value* data structure works? You have already worked quite a bit with these, as they are prominent in most programming languages. These are valuable because even when containing a large number of key-value pairs, they can "instantly" retrieve values. How can it do this, even when highly loaded?

This chapter we investigate a new data structure - one used "under the covers" to construct the collection of unordered key-value pairs known in PHP as associative arrays, in Python / Swift / C# as dictionaries, in JS as objects (minus methods, prototypes, etc.), and in C++ STL as maps. Ruby and Java have the most appropriate name for this unordered key-value data structure: Ruby calls them hashes, and Java calls them hashtables. Why? Because a *hash* function gives this data structure its quick-check, quick-retrieval feature, even when containing lots of data.

Consider the *array* data structure, which is quick-retrieval. Every array element can be immediately reached with a single index dereference. *If you know the index*, you can directly access its value: **`arr[idx]`**. This strength is also its main weakness: you *must* know index in order to access element.

The word "associative" is used with these because they *associate* a certain key with a certain value. If we use an associative array to track a specific user, we might have this: **`{ name: "Marino", age: 27, IQ: 144, languages: ['Italian', 'English'], height: 181 }`**. Here, we directly access the user's age (for example) by referencing the key: **`myUser['age']`** or **`myUser.age`**. If associative arrays didn't exist yet, how would we construct them using only traditional (numerical) arrays?

Traditional arrays associate *numerical* indices with values. The index is a key. Continuing our example to store user information in an array **`["Marino",27,144,['Italian','English'],181]`**, we can quickly access user age (27) or name ("Marino"), because we know the one and only one place in the array where we always find user age (at index **`[1]`**) or name (at index **`[0]`**). We get the benefit of quick-retrieval only if 1) for each piece of information, we have a specific index where we always store it, and 2) we remember that decision (e.g. that **`[0]`** corresponds to name, **`[1]`** to age, etc). Can we make this automatic, while retaining quick-retrieval? Yes. A *hash function* can automatically pick indices for us.

Hash functions take inputs (generally strings) and generate large, seemingly random (but repeatable) numbers, called *hash codes*. To generate a unique index for each key, we use its hash code as the index – this way each key has a reproducible index in our array where its value will be stored. Note: hash codes could be huge (or negative). To fit into our array, we limit them to a manageable range.

We solve this by constraining our array to a certain capacity, and *modulo*ing the hash codes so that they fit into that range. To store the key/value **`{ name: "Marino" }`** into our 'map', we get the hash code of the key 'name', **`mod`** that hash code to get an index that fits within the capacity of our array, and save "Marino" at that index. To retrieve the value for key 'name', we first hash the key, **`mod`** the hash

code to get an index within the bounds of our array, then retrieve the value at that index. We can store vast numbers of key/value pairs and still quickly retrieve values, without iterating through keys or values ***or*** having to remember which index corresponds to which key. It's a beautiful thing.

Chapter 13 – Hashes

We now know all we need to build the associative array data structure, also called *unordered_map*. **Map**, because the keys map to values (if it had single values, not K-V pairs, we would call it a *set*). **Unordered**, because (unlike BST or Queue) we do not know the elements' order or sequence. The hash function is sufficiently random that when we hash a certain key to a specific index (or *bucket*) in our array, we know nothing about keys that are hashed to the previous or subsequent buckets.

As with any data structure, after creating the simple constructor (at right: **`hashMap()`**), we then construct methods for pushing data into the data structure, and for checking whether a piece of data is contained in the structure. Specifically, we will create methods **`add(key, value)`** and **`contains(key)`** to do this. We also create method **`isEmpty()`**, which may suggest we add **`numKeys`** to our constructor.

Our challenges use these reference definitions:

```
function hashMap(cap) {
  this.capacity = cap;
  this.table = [];
}

//     Use the below to hash a string:
var myHashCode = myString.hashCode()

String.prototype.hashCode = function() {
  var hash = 0;
  if (this.length == 0) return hash;
  for (i = 0; i < this.length; i++) {
    char = this.charCodeAt(i);
    hash = ((hash<<5)-hash)+char;
    hash &= hash; //Convert to 32b int
  }
  return hash;
}
```

JavaScript's % operator doesn't do what we want for negative inputs (try it and see!), so we've also created a custom **`mod()`** function.

```
//     Use it this way:
var myIdx = mod(myHashCode,arrSize);

function mod(input, div)
{ return (input % div + div) % div; }
```

Add

Create an **`add(key, val)`** method on **`hashMap`** to add a new key and value to the map. This entails hashing key, mod'ing it into the size of your array, and placing the value there.

Second: what if two values hash to the same index, causing a *hash collision*? Arrays or sLists may work better than overwriting (losing) vals. Is this necessary if you have a set, not a multiset?

Is Empty

Return whether map is empty. This one-liner will require changes elsewhere.

Find Key

Create a **`find(key)`** method to return value for given key. If key is not found, return **`null`**.

Second: if you altered **`add(key,val)`** to handle collisions, extend **`find(key)`** accordingly.

Chapter 13 – Hashes

The *hash* data structure that we have built so far can add a key-value pair, retrieve the value for a key, and indicate whether it is empty. Of our classic data structure methods, these correspond to *add*, *contains* and *isEmpty*. What about the others (remove, size, front)?

First, *front* has no meaning in a key-value data structure. We don't keep the keys or values in any real order, other than how our hash code handles them – and this is not an order that we should expose to the user, since we might change our hash algorithm someday. Second, *size* is only partially relevant in a key-value data structure. A hash is similar to a circular queue in that it has a *capacity* (number of available buckets) which doesn't change, as well as some number of elements that have been added. Unlike a cirQueue, however, a hash can have more than one element map to the same bucket. Accordingly, our measure of "full-ness" is not simply the number of elements, but rather the ratio of elements to available buckets. We will call this the *load factor*. Finally, the *remove* method is exactly as you might expect: it accepts a key, and if that key is present, it removes the key-value from the data structure and returns the value.

Today we will create these exact methods and add them to our hashMap class implementation:

Remove

Create a hashMap method remove(key) that removes a key/value and returns the value (or **null** if key not found in map).

Load Factor

Once there are numerous hash collisions, we will want to grow our array size. Create a hashMap method loadFactor() to return the elements-to-buckets ratio to monitor this.

Grow

Write a method grow() to increase the internal array of buckets by 50% (20-element array would become 30 elements). Afterward, rehash all keys, since your **mod** factor has changed....

Set Size

Write a method setSize(newCap) to set the capacity of the internal bucket array to a specific length. As with grow(), after changing the array length, you must rehash all keys.

Chapter 13 – Hashes

Hash tables are sometimes referred to in classical algorithm texts as ***unordered maps***. In a map, there can be duplicate *values* (if different keys map to the same value), but there can be no duplicate *keys*. To store multiple equivalent keys one needs a *multimap*: a map that allows duplicate keys. Today you will make the changes needed to convert a hashMap into a hashMultiMap (or *unordered multimap*).

Making Maps Into Sets or Multimaps

We have previously mentioned both maps and sets. Sets are unordered collections of data, without any identifying label or index. You could think of a set as a map that has only keys. Today, we will change our hashMap data structure into a hashSet data structure, and even a hashMultiSet data structure.

Below are hashMap's attributes/functions. Referencing previous solutions, *which class attributes / methods need changing to create an unordered_multimap? An unordered_set? Unordered_multiset?*

Needs to be changed to create:	unordered_set	unordered_multimap	unordered_multiset
`constructor(size)`	☑	☑	☑
`this.capacity`	☐	☐	☐
`this.table`	☐	☐	☐
`this.numElements`	☐	☐	☐
`this.add(key, value)`	☑	☑	☑
`this.find(key)`	☑	☑	☑
`this.isEmpty()`	☐	☐	☐
`this.remove(key)`	☑	☑	☑
`this.loadFactor()`	☐	☐	☐
`this.grow()`	☐	☐	☐
`this.setSize(newSize)`	☑	☐	☑

With this, refactor hashMap, resulting in related classes for unordered_map and unordered_set. For multiset/multimap functionality, use a constructor 'allowMulti' flag rather than creating a separate class.

Chapter 13 – Hashes

Referencing previous solutions, which attributes or functions need changing to change *unordered_map* to *unordered_multimap*? To create an *unordered_set*? An *unordered_multiset*?

unordered_multimap (from unordered_map)
constructor:
```
this.multi = allowDupes || false;
```

`this.add(key,val)`: add "**`if (!this.multi)`**" around the check-key-already-exists loop.

`this.find(key)`: create **`results[]`**. If **`(multi)`**, push val for (dupe) keys found. Return results.

`this.remove(key)`: create **`results[]`**. If **`(multi)`** splice (>1) k/v found. Return results.

Others - no change (capacity, table, isEmpty, numElements, loadFactor, grow, setSize).

Summary of Map/Set Data Structures:

- **Maps** contain keys and values.
- **Sets** contain keys only.
- **Maps & Sets** have no duplicate keys: adding already-existing keys overwrites previous.
- **Multimaps & Multisets** allow duplicate keys: find() returns an array of values and remove() deletes all instances of that key.
- **Unordered** data structures use hashing to enable rapid retrieval, but lose any sequence or order. Methods nextVal(), prevVal(), min(), max(), top(), front() or back() are expensive for an unordered data structure and generally not seen.
- **Ordered** data structures use sequencing (stack, queue) or sorting (BST, heap) to retain order, but retrieval is impacted.

unordered_set (from unordered_map)
`this.add(key)`: keys only – if key found, don't replace val; otherwise, push *key* (not [key, val]).

`this.find(key)`: keys only – if found, return true (not val); otherwise, return false.

`this.remove(key)`: keys only – if key found, remove it and return true (not val); otherwise return false.

`this.setSize(newSize)`: keys only – when rehashing, **`add(key)`**, not **`[key, val]`**.

Others - no change (constructor, capacity, table, isEmpty, numElements, loadFactor, grow).

unordered_multiset (from *unordered_set*)
constructor:
```
this.multi = allowDupes || false;
```

`this.add(key)`: add "**`if (!this.multi)`**" around the check-key-already-exists loop.

`this.find(key)`: use **`numFound`**: if **`(multi)`**, **`numFound++`** (multiple times). Return **`numFound`**.

`this.remove(key)`: use **`numFound`**: if **`(multi)`**, splice (multiple), **`numFound++`**. Return **`numFound`**.

Others - no change (capacity, table, numElements, isEmpty, loadFactor, grow, setSize).

You have seen how interface (unordered map) is decoupled from underlying implementation (hash of arrays). In the same way that we examined *unordered* sets & maps, you could dive into *ordered* sets & maps. Ordered data structures care about *sequence*, so we could implement them with BSTs or heaps.

Chapter 13 – Hashes

Short Answer Data Structure Interview Questions

Today's challenges are about communication. How would you answer these if asked in an interview?

What is a queue? When would I use one?
What's the best way to implement a queue?
Can I implement this a different way? Why?
What is a stack? When would I use one?
What's the best way to implement a stack?
Is there such thing as a hybrid queue/stack?
For *queues*, which of: [push, pop, top, min, max, size, contains, prevVal/nextVal] are *fast*?
Ditto – which are slow? *How* slow?
If you needed queue.prevVal to be fast, would you change your underlying implementation?
Is there such thing as an 'unbalanced' queue?
Is there such thing as a 'full' queue?

Describe *unordered* data structures. Is there more than one type? Why would I use each?
How are unordered maps built in code?
What's the process for adding values to sets?
How do I remove a value from a set?
How do I check validity of unordered sets?
For *unordered sets*, which of: [add, remove, min, max, removeMin, removeMax, contains, prevVal, nextVal, size] are *fast*?
Ditto: what unordered set methods are slow? *How* slow are these?
Is there an 'unbalanced' set? A 'full' set?
Generally, how are sets and maps different?

What is a tree? Are they too complex to use?
Is there more than one kind of tree? Describe.
How are trees represented in code?
What's the process for adding a val to a tree?
How do I remove a value from a tree?
How would I check whether a tree is valid?
For *trees*, which of: [add, remove, min, max, removeMin, removeMax, contains, prevVal, nextVal, size] are *fast*?
Ditto – which of these are slow? *How* slow?
Is there such thing as an 'unbalanced' tree?
Is there such thing as a 'full' tree?

Describe *ordered* data structures. Is there more than one type? Why would I use each?
How are ordered maps represented in code?
How do I add values to an ordered set?
How do I remove a value from an ordered set?
How do I check validity of an ordered set?
For *ordered sets*, which of: [add, remove, min, max, removeMin, removeMax, contains, prevVal, nextVal, size] are *fast*?
Which of these methods are slow, for ordered sets? How slow?
Is there an 'unbalanced' ordered set? 'Full'?

What is a heap? How do they work generally?
What are the advantages of using a heap?
How would I a value to a heap?
What's the process to remove a heap value?
How can I check whether a minheap is valid?
For *heaps*, which of: [push, pop, min, max, removeMin, removeMax, contains, prevVal, nextVal, size] are *fast*?
Ditto – which of these are slow? *How* slow?
Is there such thing as an 'unbalanced' heap?
Is there such thing as a 'full' heap?

What is the difference between sList & dList?
When would I use one versus the other?
How are dLists implemented?
How do I check whether a dList is valid?
Which list methods are significantly different, between sLists and dLists? (front, back, pushFront/popFront, pushBack/popBack, min/max, contains, nextVal/prevVal, size)
Is there such thing as an 'unbalanced' dList?
Is there such thing as a 'full' dList?

Chapter 13 – Hashes

Blue Belt Exam

Today is a belt exam in Algorithms, leading to the rare, much-coveted Blue Belt.

Good luck! Remember our super suggestions, good guidance, terrific tips and perfect pearls of wisdom!

Chapter 13 – Hashes

(TBD) Where's the Bug? (hashes version)

Without peeking at previous code, how many bugs can you find in the hash-related code below?

Chapter 14 – Trees, Part II

Returning to the important Binary Search Tree data structure, we now build on top of our previous work, with second-level topics such as completeness, repair, partition, traversal and balance.

Full and Complete Trees

We have previously discussed whether a BST is balanced. A BST that is roughly balanced retains its excellent performance, whereas if it grows unbalanced, its performance can rapidly deteriorate. Two types of trees that take this to the extreme are **Complete** and **Full** trees.

Full trees are perfectly balanced. If each step further away from the root node is represented by a *layer* of nodes at that level, then every layer that exists in the BST is entirely filled with nodes. In other words, each leaf node's path to the root has the same length. As a side effect, every full BST contains a number of nodes that is one less than an integer power of two (i.e. 1, 3, 7, 15, 31, 63, etc). A *full* tree is the most strict type of balanced tree possible!

Complete trees are just like full trees, with one possible exception. In a complete tree, it is acceptable for the bottom layer to be less than entirely filled, so long as all nodes in that layer are as leftmost as possible. Completeness is a superset of fullness; that is, all full trees are also considered complete. A *complete* Binary Tree is as balanced as a tree with that number of nodes can be. Completeness is also *very* expensive to maintain, and for this reason complete BSTs are rarely used in production, since the costs of keeping a BST truly complete are *much* higher than the costs of adjusting it only when it becomes significantly unbalanced. In practice, complete binary trees are normally only seen in non-BST situations, such as the 'BTree projection' we see in the Heap data structure, where we interpret the underlying Array index positions as btNodes in a hypothetical BTree.

BST Is Full

Given a pointer to a BST object, return whether the BST is a *full* tree.

BST Is Complete

Given a BST object, return whether that BST is *complete*.

Chapter 14 – Trees, Part II

Repairing a Binary Search Tree

If it is possible with **`isValid()`** to detect whether a BST node is in an incorrect location, then it should also be possible to repair an invalid BST. Unfortunately, once we find an invalid node, we have no guarantee about the nodes below it – so our only recourse is to reinsert all of the subtree nodes (not just the one node we found to be invalid).

BST Repair

Given a potentially invalid **`BST`** object, create a standalone function **`bstRepair(bst)`** that rearranges nodes as needed to make it valid. Return **`true`** if you repaired the BST, **`false`** if this was not needed.

Repairing a More Complex Binary Search Tree

Refer to the **`BST2`** and **`btNode2`** data structures, from our earlier BST chapter. We based **`btNode2`** on **`btNode`**, simply adding a **`.parent`** pointer; a **`BST2`** is merely a **`BST`**, plus necessary code to use and maintain **`.parent`** in the **`btNode2`** objects it contains. As a result, invalid **`BST2`**s include not only those with incorrectly located nodes, but also those with defective pointer values (e.g.: a child's **`.parent`** doesn't point back, or node pointers create a loop!).

BST2 Repair Test Cases

In the challenge following this one, we will write code to detect and repair a potentially invalid **`BST2`**. But first, what *test cases* would you create to ensure that your solution detects and correctly fixes the possible error cases? For this challenge, a *test case* is a **`BST`** that you send to **`bst2Repair(bst2)`**.

BST2 Repair Implementation

Given a potentially invalid **`BST2`**, create standalone function **`bst2Repair(bst2)`** to detect whether it is invalid. If so, fix it and return **`true`** (if not, return **`false`**). Potential problems include incorrectly placed nodes, as well as incorrect pointers (**`.parent`**, **`.left`**, **`.right`**) that create loops, etc.

Chapter 14 – Trees, Part II

Sometimes it is necessary to divide a BST into two. To accomplish this, we might want to split the BST around a specific value, or we might simply approximate a value that would put around half the values on one side and around half on the other. Similar to how we might divide, or *partition*, a linked list into two separate linked lists, likewise when we partition a BST we want the result to be two different non-overlapping BSTs, where every node in the previous BST is contained in one of the resultant BSTs.

BST Partition Around Value

Create a method **`BST.partition(value)`** where a **`BST`** object partitions itself around the given value (whether or not that value is found in the tree). The **`BST`** should change itself appropriately, and return a new tree object containing all other nodes. Remember, the ranges of the two **`BST`** objects should not overlap (the **`max()`** of one should be less than the **`min()`** of the other).

BST Partition Evenly

Create a standalone function that, given a valid BST, partitions the tree evenly into two distinct BSTs. As in the previous challenge, change the given BST to become one of the resultant BSTs, and return the other. The two resultant trees should be valid and non-overlapping.

Second: if we don't pay attention to balance, the two resultant BSTs might be tall and thin. To improve performance, make both trees a bit more balanced before returning them.

Chapter 14 – Trees, Part II

Breadth-First Search

Previously we have talked about traversing a binary search tree. Whether we used pre-order, in-order, or post-order, we traversed from the root of the tree all the way to a leaf node, before backtracking. This is an example of **depth-first-search (DFS)**, in which we (starting at the root) explore as far as possible along each branch before exploring adjacent paths.

Can you think of a scenario in which DFS is not the best way to traverse a binary tree? Let's say we have a generic binary tree (not a BST) with 300 nodes, and each node contains a single upper-case letter. How would we find the 'Q' that was closest to the root node? To solve this problem, Depth-First Search would not be a great choice. What if we wanted to find ***all*** the instances of the letter 'Q'? In that case, we need to visit every node anyway, so DFS is a reasonable choice.

To find the closest 'Q', a better strategy would be, as Jerry Seinfeld advised, "go for the miracle parking spot, then concentric circles." With **breadth-first search (BFS)**, we would first start looking at the root node itself, then proceed to every possible node that is only one step further away, then advance to every node that is one step further, etc. (like concentric circles, but in a bTree).

BFS, when used in conjunction with a Queue data structure, is a good way to advance a search evenly away from the starting point. Remember, once we examine a given node, we must then check all other nodes of similar distance from the origin, before we check that given node's children. The Queue can help us remember remember a node, from the moment we examine *its parent* until the time we check *it*.

We will encounter BFS later with graphs; for now breadth-first is an important new way to iterate trees.

BSTLayerValues

Given a BST and a layer number (starting at zero for the root), return an array containing all the values at that layer in the BST.

BST2LayerArrs

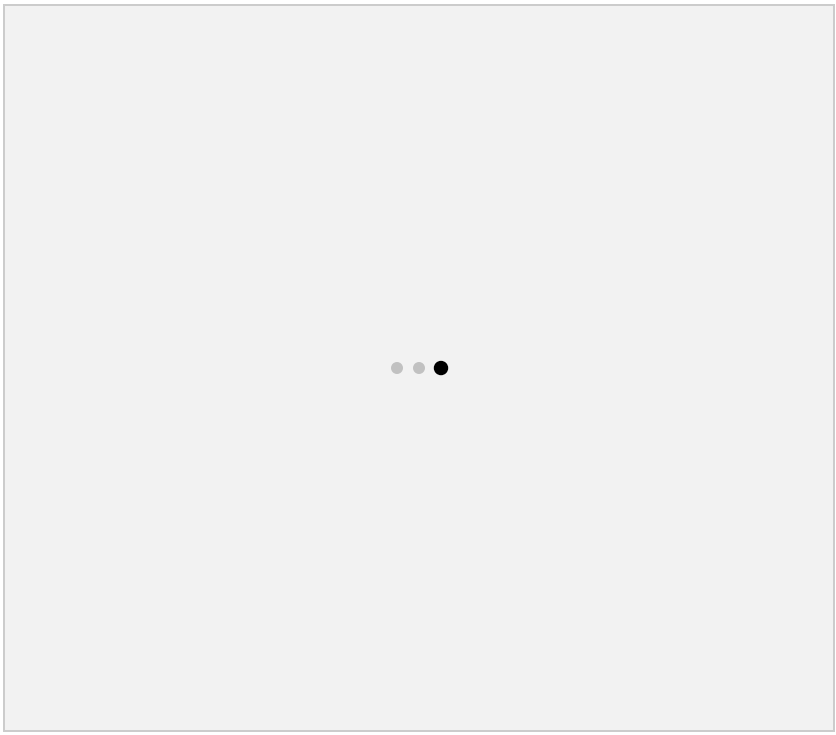
Given a BST, return a two-dimensional array containing all values in the BST. The outer array represents each layer (starting at zero for the root), and the inner array for each layer represents the values at that layer in the BST.

(TBD) Chapter 15 – Tries and Graphs

Adding to our knowledge base of reference-based data structures (lists, trees), in this chapter we learn about **tries** and **graphs**.

Trie Data Structure

The *trie* data structure (initially pronounced "*tree*" after the middle syllable of *retrieval*, but now universally pronounced "*try*" to avoid confusion) is useful in scenarios usually reserved for a hash, but it is best known for its power to predict – particularly auto-complete. Let's examine the trie *conceptually*.



As shown at left, a trie is a set of connected nodes. The root node represents the empty string ""; from that node, there can be many pointers to other nodes. Each pointer+node represents the addition of a letter to that sequence. If the sequence of letters adds up to a word, then the word is stored within the node as well. If not, it still exists in the trie if it is needed for another, longer word (like the 'R' node below the top-level 'A'). Because each node represents a unique word, two parent nodes can never point to the same child node – although 'ATE' and 'ARE' are both in our trie, the 'AR' node and the 'AT' node point to different children for an additional 'E'.

Nodes are added only when they are needed to store a word – the diagram above represents the words "a", "are", "arf", "as", "asp", "at", "ate", "atom", "atop" and "I". If we were to add the word "asps", a pointer representing "s" would extend from the "ASP" node to a new node containing "ASPS". If we add the word "ar", a node already exists at the appropriate location, so "AR" would be stored into the node.

Codewise, a **`TrieSet`** object is quite similar to a basic binary tree object: it contains a single attribute – a pointer to the root node – plus the instance methods. Each **`TrieNode`** object contains an optional node value, plus an array of pointers. This array is initially empty but can hold as many pointers as there are letters in the alphabet. If we choose to lower-case all the strings that we hold in our **`TrieSet`**, then each node's array could contain as many as 26 pointers. Again, each pointer+node represents the addition of a specific letter to the sequence of letters from the root up to that node.

To add a word to our trie, we iterate each letter, following the appropriate pointer for that letter. If the node's array has no pointer there, we create a node, write it to the array location and proceed. When we reach the last letter, we save the word in the node, signifying that it is *terminal*, not intermediate.

Chapter 15 – Tries and Graphs

After walking through a trie in both concept and code, hopefully you see how tries might enable one to predict the word, given the first few letters! Implement the following challenges to create a **`TrieSet`**:

Trie Insert

After building simple **`TrieSet`** and **`TrieNode`** constructors, create an **`Add`** method to insert a string to the set. For our purposes, assume that input strings will be letters only – no numerals or punctuation. Also, you can convert inputs to lowercase before storing them. Return **`false`** if word has already been stored (after all, it's a **`TrieSet`** not a **`TrieMultiSet`**!), or **`true`** if insertion is successful. Tries are treelike (not a purely linear one like a linked list), so recursion at the node level is a reasonable choice.

Trie Contains

Create a **`TrieSet`** method to check whether given string is present within the set. Again, assume letters-only inputs; you can convert all strings to lowercase. Return **`true`** if word is found, **`false`** if not.

Trie First

Tries are reasonable substitutes for Hashes, yet they retain order! Build a method to return our trie's first value. Here, *first* means lowest-alphabetically, not earliest-added. N.B.: "ab" comes before "aba".

Trie Last

Create a method to return the trie's *last* (highest-alphabetical) value. Note: "zazzy" comes after "zaz".

Trie Remove

Construct a method in the **`TrieSet`** class that removes the given string from our set. As earlier, you can safely assume that all input strings will contain only letters (not numerals or punctuation). Also, you can safely lowercase all strings before storing or checking for them. When removing a trie value, remember that in some cases you need to remove the terminal **`TrieNode`**. Furthermore, you might need to remove certain ancestor intermediate nodes. Make sure to return **`true`** if the removal was successful, and **`false`** if given string was not found.

Second: Incorporate punctuation and case-sensitivity across the entire **`Trieset`** class. Suggestion: the 95 typeable characters on a keyboard have consecutive charCode values, starting with [space].

Chapter 15 – Tries and Graphs

Today we finish our TrieSet implementation, including auto-complete – a very cool feature used by every search engine and mobile device.

Trie Size

Return the number of values added to the trie. There are two valid ways to implement this method – can you come up with both? In which usage cases would you prefer one over the other?

Trie Next

Given a string that might or might not be found in the trie, return the contained string that is immediately subsequent. Hashes don't do well with this, but tries can! Return `null` if there is no subsequent string.

Trie Auto Complete

Assume that your trie has been populated with a wide array of valid words. Given a string (presumably what a user has typed so far), use your trie to rapidly produce and return an array of the words that begin with that string.

Second: augment `autoComplete(str)` to accept `maxResults`; return at most that many results.

Chapter 15 – Tries and Graphs

Trie MultiSet

We mentioned it earlier, so let's build **`TrieMultiset`**. Remember, multiset is identical to set, except it tracks number of instances along with value. Implement the following that build a **`trieMap`**:

Trie MultiSet Insert

For this exercise, **`Insert`** increments count, adds nodes as needed and always succeeds.

Trie MultiSet Remove

`Remove` should decrement count but return *previous* count (0 if not found) and eliminate nodes as needed.

Trie MultiSet Contains

`Contains` should return count (0 if not found).

Trie MultiSet Size

`Size` should return total (multi) count.

Note: **`First`** / **`Last`** / **`Next`** are unchanged.

Trie MultiSet Auto Complete

Given an entire dictionary and a short initial string fragment, autocomplete might return a huge number of results. Let's use the **`count`** aspect of each **`TrieMultiNode`** to denote the frequency of that word, and use this to prioritize the return results from autocomplete, so that most frequent words are listed first. In addition to the string fragment, accept **`maxResults`**, and return at most that many results.

TrieMap

We can expand a **`TrieSet`** to a **`TrieMap`** by associating an additional value to each stored string.

TrieMap Insert

`Insert` should accept a key and a value, and should return the preexisting value (if key already existed) or **`null`** if key is new.

TrieMap Contains

`Contains` should return the value for the given key (**`null`** if key is not found).

TrieMap Remove

`Remove` deletes the given key (and value), returning **`true`** if key was found, else **`false`**.

TrieMap Size

`Size` is unchanged from **`TrieSet`**.

TrieMap First

`First` returns an object containing the key-value pair for the alphabetically-lowest key.

TrieMap Last

Conversely, **`Last`** returns the final key-value.

TrieMap Next

Given a key that may not be present, **`Next`** returns the subsequent key-value.

Chapter 15 – Tries and Graphs

(TBD) Graphs

Why use a Graph? How the graph is used. Different ways that graphs are represented.

(TBD) Adjacency List

(TBD) Adjacency Map

(optional)

Chapter 15 – Tries and Graphs

(TBD) Depth-First Search

When to use DFS? When not to use DFS?

(TBD) Path Exists

(TBD) Someone on the Inside

(TBD) Breadth-First Search

When to use BFS? When not to use BFS?

(TBD) Shortest Path

(TBD) Graphs: Seven Degrees of Kevin Bacon

Chapter 15 – Tries and Graphs

(TBD) Directed Acyclic Graph

What Directed means? What Acyclic means? Different things that are represented by DAGs. What you can do with a DAG that you can't do with other types of graphs.

(TBD) Is DAG

Given a graph, determine whether it is a DAG.

(TBD) DAG2Arr

Given a Directed Acyclic Graph, create an array where every node

(TBD) Word Ladder

(http://www-01.sil.org/linguistics/wordlists/english/wordlist/wordsEn.txt)

Chapter 15 – Tries and Graphs

(TBD) Where's the Bug? (trie/graph version)

Without peeking at previous code, how many bugs can you find in the trie/graph-related code below?

(TBD) Chapter 16 – Optimization and Estimation

(TBD) Deck Of Cards

OOP: deckOfCards

Chapter 16 – Optimization and Estimation

(TBD) Speed Prime

(various optimizations)

Chapter 16 – Optimization and Estimation

(TBD) Optimizing N-Queens

various additions and optimizations

Chapter 16 – Optimization and Estimation

Estimation

Estimation problems exist generally to assess a candidate's ability to break a problem into smaller pieces that can be solved separately. In some cases these problems can also measure a person's judgment. The trick here is to not get hung up on determining a precise answer – keep in mind that each component in your solution is likely only accurate to an order of magnitude. With this in mind, use numbers that will make the mental math easier for yourself.

Piano Tuners

How many piano tuners are there in the United States?

Gas Stations

How many gas stations are there in the state of California?

Kindergarten Teachers

How many kindergarten teachers work in the state of Washington?

Earth's Circumference

What is the circumference of the earth?

Weight of a Ferry

What is the weight of the Bainbridge Island ferry? Full or empty?

Basketballs in a 747

How many basketballs would fit into a 747? With your answer, state which parts of a 747….

Chapter 16 – Optimization and Estimation

Black Belt Exam

Today is a belt exam in Algorithms, leading to the extremely rare, very-much-coveted Black Belt in advanced Algorithms & Data Structures.

Good luck! Remember our super suggestions, good guidance, terrific tips and perfect pearls of wisdom!

Chapter 17 – Bit Arithmetic

Numerical Systems

As humans we have been raised from a young age to have a specific attitude toward the number ‘ten’. This is rooted in the fact that as a species we have ten fingers, ten toes. As a result, there are ten numerals in the modern symbolic representation of numbers (0-9). But how would our world look if from the beginning our species had only eight fingers? Or what if we had *sixteen* fingers? Or what if we, like computers, primarily thought about only “on” and “off” – the equivalent of having only two numerals to choose from? Our writing would look different, although the actual quantities would be the same.

Numerical systems by nature are simply different ways that symbols can represent a quantity. A certain quantity can be represented by different symbols. Whether we say “quarante-deux” or “forty-two” or 0x2A or ***42***, the amount is the same. So, why can’t we just stay with the *decimal* (ten-based) system that is ‘native’ and natural to humans?

Computers don’t think that way, that’s why. Computer architecture from the beginning has evolved from a foundation based on binary digital logic, where signals are either ON or OFF. They live in a world of two numerals, not ten numerals – almost as if they count using two fingers, rather than ten fingers. Each of these 0/1 numerals is called a *bit*. If you can only use 0’s and 1’s then a number like 42 (decimal) becomes 0b0101010 (binary). This would make for very long and tedious numbers indeed, except that we have combined these groups of 2, to create number systems that they are closer to the 10-based system that is natural to us.

Chapter 17 – Bit Arithmetic

Octal System

There is a number system called 'octal' that uses 8 numerals, grouping three bits together into one numeral that goes from 0 to 7. After counting up to 7, we start using an additional digit, just like in our usual decimal system we start using the "tens" digit when counting beyond 9. The indicator that we are using octal is the prefix '0o', so the number 8 (decimal) will appear in octal as 0o10. In decimal notation each numeral has a jump in value of 10x as we move from the 'ones' digit to the 'tens' digit, or from the 'tens' digit to the 'hundreds' digit. Similarly in octal each digit represents a jump of 8x. A number like 0o4213, then, is $(4 \; x \; 8^3) + (2 \; x \; 8^2) + (1 \; x \; 8^1) + (3 \; x \; 8^0) = 2187$ (decimal). You won't need to deal with octal numbers much – if anything, you will deal with either base-2 or base-16 – but it gives a good introduction to the idea that computers don't have ten fingers and hence don't agree with our completely arbitrary decision to think about numbers as having exactly 10 numerals.

Decimal to Octal Practice

For practice, convert the following from decimal to octal representation. Example: 31 becomes 0o37.

13 6 25 8 45 10 -9 64 255

Octal to Decimal Practice

For practice, convert the following from octal to decimal. Example: 0o47 becomes 39.

0o610 0o5 0o26 0o47 0o302 0o0 -0o12 0o76 0o101

Decimal to Octal String

Create a function **`dec2OctStr(value)`** that converts a number into a string representing that number in octal notation. For this challenge, do not use the (very convenient) **`toString`** function.

Octal String to Value

Create a function **`octStr2Val(str)`** that accepts a string representing an integer in octal notation, and returns the value. For this challenge, do not use the (very convenient) **`parseInt`** function.

Chapter 17 – Bit Arithmetic

Octal represents groups of three bits, and since three is an odd number, it isn't the very best way to represent how a computer natively thinks about numbers. It is much more common for us to represent numbers in groups of four bits (hexadecimal) or simply one bit at a time (binary).

Hexadecimal System

For a second, imagine that each of us have had eight fingers on each hand, since birth. As a result, when we created our numeral system, we created more than just 0-9 numerals. Instead, we created enough numerals to represent every quantity that was "less than two hands" worth. In our hypothetical world, that means there are extra numerals after 9 that represent *in a single character* the amounts up to "two hands worth". That system is hexadecimal (16-based), and hexadecimal numbers are prefixed by 0x ("zero-X"), just as octal numbers are prefixed by 0o ("zero-Oh"). The extra numerals 0xA, 0xB, 0xC, 0xD, 0xE, 0xF are equivalent to 10, 11, 12, 13, 14, 15. Each additional digit in hexadecimal is a multiplication by 16, so the number 0x10 is equivalent to 16. The number 0x2A is equivalent to 42.

Decimal to Hexadecimal

For practice, convert the following from decimal to hexadecimal. Example: 31 becomes 0x1F.

13 6 25 8 45 10 -9 64 255

Hexadecimal to Decimal

For practice, convert the following from hexadecimal to decimal. Example: 0x47 becomes 71.

0xDB 0x5 0x20C 0x4F 0xB2 0x0 -0x12 0x7E 0x101

Decimal to Hexadecimal String

Create a function **`dec2HexStr(value)`** that converts a number into a string representing that number in hexadecimal notation. For this challenge, do not use the (very convenient) **`toString`** function. For example, given the value 108, the function should return '0x6C'.

Hexadecimal String to Value

Create a function **`hexStr2Val(str)`** that accepts a string representing an int in hexadecimal notation, and returns the value. For this challenge, do not use the (very convenient) **`parseInt`** function. For example, given the string '0x1D2', the function should return 466.

Chapter 17 – Bit Arithmetic

Binary System

If “hex” numbers make sense to you, then good job – you are starting to think like a computer. If you *really* want to get geeky, you’ll need to understand binary as well. In binary each additional digit is an additional power of 2, so a number like 0b1111111 = $(1 \; x \; 2^7) + (1 \; x \; 2^6) + (1 \; x \; 2^5) + (1 \; x \; 2^4) + (1 \; x \; 2^3) + (1 \; x \; 2^2) + (1 \; x \; 2^1) + (1 \; x \; 2^0) = 255$, or 0xFF. Each four bits of binary translate into a single hex digit, so that translation should be fast. 0x3 = 0b0011, 0x8 = 0b1000, 0xB = 0b1011, 0xE = 0x1110.

Decimal to Binary

For practice, convert the following from decimal to binary. Example: 117 becomes 0b1110101.

13	6	25	8	45	10	-9
64	255	128	35	0	198	

Binary to Decimal

For practice, convert the following from binary to decimal. Example: 0b100111 becomes 39.

0b10100101	0b111	0b1111000	0b110110	0b000	0b100101	0b11
-0b1010	0b100110	0b1010101010				

Decimal to Binary String

Create a function **`dec2BinStr(value)`** that converts a number into a string representing that number in binary notation. For this challenge, do not use the (very convenient) **`toString`** function. For example, given the value 35, the function should return ‘0b100011’.

Binary String to Value

Create a function **`binStr2Val(str)`** that accepts a string representing an int in binary notation, and returns the value. For this challenge, do not use the (very convenient) **`parseInt`** function. For example, given the string ‘0b1010101’, the function should return 85.

Chapter 17 – Bit Arithmetic

Bitwise Operators, Part 1

Most math operators don't know or care about a number system. Addition is addition, whether "10 + 11 = 21" or "0b1010 + 0b1011 = 0b10101". Ditto subtraction, multiplication, division, negation, comparison and equality. Other operators make sense only if we think of numbers in binary representation. To best understand these *bitwise operators*, we must first consider three logical operators (AND, OR, NOT). These operators work on Boolean values, which are essentially one-bit values (true = 1, false = 0).

- **&&** (logical AND) operates on two booleans. It returns **`true`** only if both are **`true`**, else **`false`**.
- **||** (logical OR) operates on two booleans, returning **`true`** if either is **`true`**, otherwise **`false`**.
- **!** (logical NOT) operates on a single boolean value, inverting **`true`** to **`false`** and vice versa.

Bitwise operators are equivalent to logical operators, except they work one-bit-at-a-time across entire numbers. Some bitwise operators accept two numbers, and others work on a single number.

- The **bitwise AND** operator is **&**. It operates on two numbers and compares them, one bit at a time (their least-significant bits, their second-least-significant bits, etc). If both bits are 1, then that same bit in the result is set to 1, otherwise that bit is set to 0. It does this for every bit in the two operands. Example: 0b10101010 & 0b01100110 = 0b00100010.
- The **bitwise OR** operator is **|**. It operates on two numbers and compares them, one bit at a time (least-significant bits, second-least-significant bits, etc). If either are 1, then that bit in the result is set to 1, otherwise it is set to 0. It does this for every bit in the two operands. Example: 0b10101010 | 0b01100110 = 11101110.
- The **bitwise NOT** operator is **~**. It operates on one number, examining one bit at a time in isolation. Each bit in the original is inverted in the result. Each 1 becomes 0; each 0 becomes 1. Example: ~0b10101010 = 0b01010101.

Note: JavaScript stores numbers as 64-bit (floating-point) values, but its bitwise operators operate on 32-bit integers. So before any bitwise operation begins, values are converted to 32-bit integer format.

For the expressions below, indicate the results in binary, hexadecimal and decimal notations:

Bitwise AND

```
0b010101 & 0b0110111   57 & 87   0b01101001 & 0b00011000   0xBABE & 0xBEEF
```

Bitwise OR

```
0b010101 | 0b0110111   57 | 87   0b01101001 | 0b00011000   0xBABE | 0xBEEF
```

Bitwise NOT

```
~0b010101  ~0b0110111   ~5787   ~0b01101001  ~0b00011000  ~0xBABE  ~0xBEEF
```

Chapter 17 – Bit Arthmetic

Bitwise Operators, Part 2

Having studied & (bitwise AND), | (bitwise OR), and ~ (bitwise NOT), let us continue to the other important bitwise operations.

- The **bitwise XOR** (exclusive or) operator is **^**. It operates on two numbers and compares them, one bit at a time. If the bits are different than each other, then that bit in the result is set to 1, otherwise it is set to 0. It does this for each bit in the two operands. Example: 0b10101010 ^ 0b01100110 = 0b11001100.
- The **bitwise LSL** (shift left) operator is **<<**. It operates on two numbers and shifts the bits in the first number to the left; the second number indicates the number of places by which to shift the number. Numbers are treated as 32-bit integers; with each shift the most-significant bit (bit 31) is lost, and a value of 0 shifts into the least-significant binary digit (bit 0). Example: 0b111101110000000011110000110010100 << 3 = 0b101110000000011110000110010100000.
- The **bitwise LSR** (logical shift right) operator is **>>>** (yes, three > symbols not two). It operates on two numbers, shifting the first number to the right by the number of bits indicated by the second number. Numbers are treated as 32-bit ints; with each shift the number loses its least-significant bit (bit 0), and 0 shifts into the most-significant binary digit (bit 31). Example: 0b10111111000000001111000011001010 >>> 3 = 0b000101111111000000001111000011001.
    - JavaScript's **ASR** (arithmetic shift right) operator **>>** is identical to >>> in every way except that when shifting right, the most-significant binary digit (bit 31) is retained as the most-significant binary digit, instead of a zero being added. For our purposes, >>> is much more useful than >>. Example: 0b11111111000000001111000011001010 >> 3 = 0b11111111111000000001111000011001.

For the operations below, indicate the results in binary, hexadecimal and decimal notations:

Bitwise XOR

```
0b010101 ^ 0b0110111       57 ^ 87          0b01101001 ^ 0b00011000
         0x0BADCACA ^ 0xD00DAD    0xCAFED00D ^ 0xDECAF    123 ^ 124
```

Bitwise LSL

```
0b010101 << 7              57 << 8          0b01101001 << 0b00000111
           0xF00D << 0xA         0x000BABEE << 0b1       42 << 0xA
```

Bitwise LSR

```
0b0101010101 >>> 7        157 >>> 3         0b10110100101010011 >>> 15
          0b00011000 >>> 2       0xDEADBEEF >>> 0xA       0xCAFEBABE >>> 0b11
```

Chapter 17 – Bit Arthmetic

Bit Shifting and Masking

To save space, we can encode small numbers into larger containers. We just need to use our shift operators to move the small values into a different section of the container. For example, the number 0x1CED can be left-shifted by sixteen (`<< 16`) to 0x1CED0000, leaving space for another 16-bit num. We would combine these values using `|`, such as: `(0x1CED << 16) | 0xF00D == 0x1CEDF00D`.

Sometimes bits in a number represent 32 independent ON/OFF states (such as the Yes/No votes of 32 people, or the current settings of 32 light switches in your house). This can be much more convenient than storing each of the 32 values individually, as long as we can manipulate each bit separately.

Thinking about our bitwise operators, every bit can be set by using a combination of `|` and **<<**. To set bits 17 and 18 but keep the rest of the variable untouched, we could `val |= (0x3 << 17)`. Those bits might have previously been 0 or 1, but now they are 1 (because any value **OR 1** becomes 1).

On the flip side, we may want to clear a certain bit. In that case we take advantage of the fact that any value **AND 0** becomes 0. If we don't want to affect the rest of the variable when we clear that bit, then we must **AND** it with a very specific *bit mask*. This bit mask should have a value of all-ones, except for the one bit we want cleared. The **~** operator flips every bit value, making bit masks easy to create. To clear bit 12, but keep the rest of the variable untouched, we could `val &= ~(1 << 12)`. The `(1 << 12)` creates a *positive mask* so to speak, and the **~** turns it into a negative mask. Our variable might previously have had a 0 or a 1 at bit 12, but now it is 0.

Using what we've learned earlier, solve these bit-related challenges today:

CountSetBits

Given integer, return how many of its bits are set to 1. Given 1023, return 10. Given 8192, return 1. **Second:** make it O(s): s is number of *set* bits.

EncodeBytesTo32

Given four values between 0-255, encode them into a 32-bit integer. First should map to most significant 8 bits. Given [0xF0, 0xC3, 0x96, 0x59], return 4039349849 (0xF0C39659).

Decode32ToBytes

Given a 32-bit integer, decode and return a set of values corresponding to the four bytes in the integer. Example: given 306542763 (which in hex is 0x124578AB), return [0x12, 0x45, 0x78, 0xAB].

ByteArray

With encode/decode you've written above, create a ByteArray data structure to store 8-bit values encoded into 32-bit ints to save space. Build **`set(index, value)`** and **`get(index)`**.

Chapter 17 – Bit Arthmetic

Here are end-of-chapter challenges to put your new low-level nuts-and-bolts skills to work:

EncodeBitNum

Given a bit value, bit number 0-31, and existing 32-bit value, mask the bit into the 32-bit value and return its new value. For example, given 1, 30, 1, return 0x40000001. Given 0, 3, 0x18, return 0x10.

DecodeBitNum

Given a bit number and an existing 32-bit number, decode and return the bit number referenced. For example, given 30, 0x4FFFFFFF, return 1. Given 3, 0x4FFFFFF7, return 0.

BitArray

With encode/decode functions you've written above, create a BitArray class that stores 1-bit values encoded into 32-bit integers. Include methods **`set(index, value)`** & **`get(index)`**.

RadixSort2

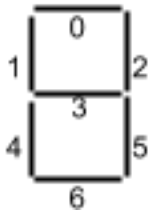
Implement RadixSort, based on powers of two instead of digit numerals 0-9. Sort by lowest significant bit, then by next least significant bit, etc. What is the big-O runtime to sort 32-bit integers?

Sprinklers

The Rockefellers' country estate is watered by a 28-section sprinkler system. Create a function that returns a 28-bit number; the landscape microcontroller will call this function each minute to determine which sections to run. Only one sprinkler should run at any time, and each should run 20 minutes per day. Four global variables alter the system's behavior, in increasing priority: RAIN_SENSOR represents the precipitation meter – if this variable is true, disable all sprinklers. SENSOR_OVERRIDE, if true, disables the precipitation meter. While SYSTEM_TEST is true, cycle one at a time through all 28 sprinklers for a minute each. Finally, if MASTER_SYSTEM_DISABLE is true, turn off all sprinklers.

LED Encoding

Classic LEDs have segments that are individually turned on or off to produce each letter/numeral, arranged as in the diagram at right. The on/off state of each segment is determined by a different bit in a container byte (numbered at right). A value of 0x7B equates to LED segments [0, 1, 3, 4, 5, 6] enabled, displaying numeral '6'.



Create function **`LED2Numeral(ledByte)`** that accepts a byte representing the combined states of LED segments in one base-10 numeral, and returns the value of that base-10 numeral (i.e. 0-9). For example, given 36 (0x24, or LED segments 2 & 5), this function should return 1.

Second: create function **`Int2LED(value)`** that accepts a 16-bit integer value and translates it into the values needed to produce the corresponding LED readout in base-10. The function should return an

array of five bytes: each byte representing one of the numerals from *least-significant* to *most-significant*. Using our examples above, **`LEDBytes(85210) == [0, 36, 93, 107 127]`**.

Chapter 18 – Trees, Part III

AVL Trees

Remember that a binary search tree's performance is linked to how balanced it is. If a tree is unbalanced and deep, then there is a chance that the value we seek is down in the depths of the tree, far beyond the average expected height. But what if our BST could somehow keep itself balanced? How would it do this, and how expensive would it be? Soviet mathematicians Georgy Adelson-Velsky and Evgenii Landis responded to this problem by inventing the first self-balancing tree: named after their surname initials, we call it the **AVL tree**.

The rules of an AVL tree are simple: for every node, the heights of its two child subtrees must differ by at most one. If an insertion or removal changes the tree so that this rule is no longer valid (in other words, an insertion or removal makes the tree *unbalanced*), the tree must 'rotate' its shape to become balanced again. To optimize the AVL tree for the fact that it will constantly check its balance at various locations, each node contains (and maintains) a *balance factor* (1 if its left subtree is one node deeper than its right subtree, -2 if the right subtree is deeper by two, 0 if both sides are even, etc). For today, let's just measure and detect these situations; tomorrow we will address them.

```
function AVLTree() {
  var head = null;

  this.add = function(value) {}
  this.remove = function(value){}
  // Assume these exist and
  // correctly update node.balance

  this.height = function() {
    // ...write this code today
  }
  this.isBalanced = function() {
    // ...write this code today
  }
}
```

```
function AVLNode(value) {
  this.val = value;

  this.balance = 0;
  this.left = null;
  this.right = null;

  this.height = function() {
    // ...write this code today
  }
  this.isBalanced = function() {
    // ...write this code today
  }
}
```

Height (AVL)

Given an AVL tree, create **`height()`** methods for AVLTree and AVLNode objects.

Is Balanced (AVL)

Given an AVL tree with up-to-date balance values, create **`isBalanced()`** methods for the AVLTree & AVLNode classes.

Chapter 18 – Trees, Part III

AVL trees are a type of *self-balancing tree* (we touch on other variants later). The motivation for staying balanced is efficiency: unbalanced trees do not add/remove/find as quickly. However, keeping balanced might be expensive; if we aren't careful, the costs eclipse the benefits. How can we minimize costs?

1. Minimize the cost of checking a tree's cost. This implies that we:
    a. Store a value in each node, rather than recomputing height/balance each time;
    b. Store *balance,* not height, to avoid checking children when testing for balance;
    c. Check the tree's balance only at appropriate times;
    d. Check the tree's balance only at necessary tree locations.
2. Minimize the cost of maintaining the tree's balance indices, implying that we:
    a. Only update the balance indices when we add/remove values (or rebalance);
    b. Update balance indices for *only nodes affected* by the add/remove, so that we minimize the number of nodes whose balance need rechecking (see 1c).
3. Minimize the cost of rebalancing the tree, when this is needed. This implies that we:
    a. Minimize the number of nodes changed during a rebalance, so that in turn we
    b. Need to update *balance* for only a small number of nodes (see 2b), in order to minimize the number of nodes whose balance needs rechecking (see 1c).

What does 1b mean? **`AVLNode.isBalanced()`** is inexpensive if we maintain **`.balance`** – it's a quick attribute check. If instead we store **`.height`**, **`isBalanced()`** requires checking and comparing heights of both children. This makes *updating* balance more tricky but contributes to a successful item 1 in our list. Method **`height()`** is less critical, but **`.balance`** tells us which way to branch as we dive to the deepest leaf. It isn't O(1), but it is *O(NlogN)* which is good enough.

Let's explore 2a and 2b. Remember those methods that yesterday we glossed over, *assuming* they already existed? We will create them today. Write **`add(value)`** and **`remove(value)`** methods for the AVLTree class. When you do so, remember to keep the **`.balance`** attribute up-to-date for each node.

```
function AVLTree() {
   var head = null;
// assume isBalanced() works fine
   this.isBalanced = function() {}

// ...write these today
   this.add = function(value) {}
   this.remove = function(value){}
}
```

```
function AVLNode(value) {
  this.val = value;
  this.balance = 0;
  this.left = null;
  this.right = null;

// assume isBalanced() works fine
   this.isBalanced = function() {}
}
```

AVL Add

Create **`add(value)`** for the AVLTree class. Update **`.balance`** for any affected nodes, but don't worry about rebalancing the tree.

AVL Remove

Create **`remove(value)`** for the AVLTree class. Update **`.balance`** for any affected nodes, but don't worry about rebalancing the tree.

Chapter 18 – Trees, Part III

Let's review the challenges we face, if we want an AVL tree to have high performance:

1. Minimize the cost of checking a tree's cost. This implies that we:
    a. Store a value in each node, rather than recomputing height/balance each time;
    b. Store *balance,* not height, to avoid checking children when testing for balance;
    c. Check the tree's balance only at appropriate times;
    d. Check the tree's balance only at necessary tree locations.
2. Minimize the cost of maintaining the tree's balance indices, implying that we:
    a. Only update the balance indices when we add/remove values (or rebalance);
    b. Update balance indices for *only nodes affected* by the add/remove, so that we minimize the number of nodes whose balance need rechecking (see 1c).
3. Minimize the cost of rebalancing the tree, when this is needed. This implies that we:
    a. Minimize the number of nodes changed during a rebalance, so that in turn we
    b. Need to update *balance* for only a small number of nodes (see 2b), in order to minimize the number of nodes whose balance needs rechecking (see 1c).

For 2a and 2b, as you discovered yesterday, we minimize the cost of updating balance indicess by only updating the balance of nodes being inserted/removed and their ancestors upward (*not* the entire tree). When an ancestor node's balance is unaffected, we need not continue checking upward.

What about our other major implication: rebalancing? We handle this with an operation called *rotation*.

Rotation

The benefit of AVL trees over other BSTs is that AVL trees automatically keep themselves relatively balanced. When an AVL tree discovers an imbalance (if any node's left subtree height and right subtree height differ by more than one), it fixes that condition by *rotating* that node.

Think of rotation as a clockwise (Rotate Right) or counter-clockwise (Rotate Left) shift of both the node in question as well as its "tall" child node. Child is promoted above parent, reducing overall tree height.

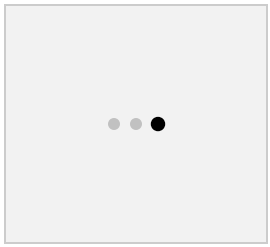
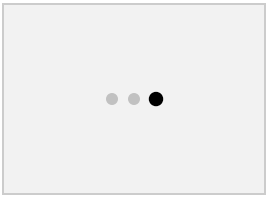
Consider a large BST where in the midst of the tree, node A has a **`.right`** child: node B. Following an insertion somewhere below B, node A's right subtree height is now two greater than its left subtree height. We should *Rotate-Left* node A. This will change the height for A and B, but how are all the other nodes affected? It would be expensive to move lots of nodes around whenever we do a rotation.

Fortunately, this isn't the case. Think about where the rest of the nodes in the tree *end up*, when we do a rotation. To start with, nodes above this rotation do not move, nor do those in other parts of the tree. As a result we only need to worry about A's and B's children. Those children have values that are either

a) less than A's (conveniently located under **`A.left`**), or
b) greater than/equal to B's (conveniently located under **`B.right`**), or
c) between A and B (*currently* conveniently located under **`B.left`**).

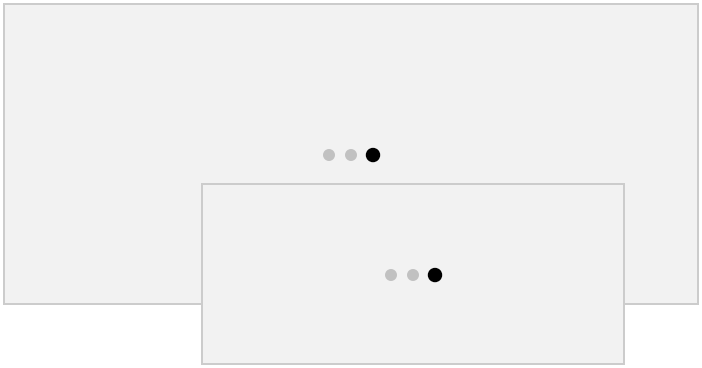
When node A is Rotate-Lefted, it becomes the **`.left`** of node B. Let's put their groups of children in place. Child nodes less than A (the entire **`A.left`** subtree) should stay where they are. Nodes greater than B (the **`B.right`** subtree) should also stay put. However, nodes *in between* might pose a problem. They can't stay where they are (under **`B.left`**), because that's where A has moved. What to do?

Let's reason through this. The nodes in question have values less than B, so they should go to B's *left* somewhere. They have values that are greater than A, so they should go to A's *right* somewhere. After promoting B, notice that **`A.right`** is now available! The subtree previously located at **`B.left`** can move to **`A.right`**. *Voila!* Our Rotate-Left is complete.

After  calling **`RotateLeft(redNode)`** on tree at left, it becomes the tree pictured at right. 

This is *super cool!* You should draw a few diagrams to prove to yourself that it works as it should. When you do this, you may find a *corner case* where the rotation changes the shape of the tree but doesn't address the imbalance (in other cases, rotation fixes the imbalance beautifully. Let's revisit the previous example, before the rotation. What if our imbalance is caused by a tall **`B.left`** subtree? That subtree shifts over to **`A.right`** during the RotateLeft; it doesn't move closer to the root; our imbalance remains. The objective of our rotation, really, is to pull this 'tall grandchild' subtree closer to the root.

View the tree below. If the **`.left`** of the *tall* right child causes imbalance, we can't simply **`RotateLeft(red)`**. We must first **`RotateRight(green)`**, transforming the first tree into the middle tree. Then, a **`RotateLeft(red)`** transforms the middle tree into a (shallower) final tree below.



Rotate Left

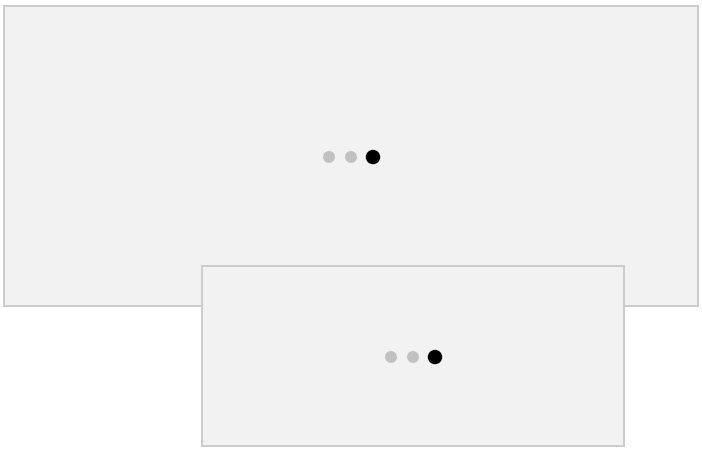
Using your new rotation cleverness, create a **`rotateLeft(node)`** method in the AVLTree class. Make sure to first counter-rotate the child if needed, and as always update **`.balance`** attributes appropriately and inexpensively.

Rotate Right

Create a **`rotateRight(node)`** method for AVLTree. Counter-rotate the ‘tall child’ first, if needed, and keep all **`.balance`** attributes appropriately and inexpensively up-to-date.

Chapter 18 – Trees, Part III

Let's review how the nodes move, on the Right-Left Rotation that we did yesterday. If the **`.left`** of a *tall* **`.right`** child causes imbalance (specifically black), then before we **`RotateLeft(red)`** we should first **`RotateRight(green)`**. This transforms the first tree into the middle tree. Then, a **`RotateLeft(red)`** transforms the middle tree into a (shallower) final tree below.



Note: the heights of various child nodes (blue, pink, orange, yellow) don't change during this rotation process; the heights of ann nodes in those subtrees are unaffected, all the way to the leaves. Heightwise, the only affected nodes are red, black, and green – *as well as their parent chain*. For this reason we must follow any height change upward to that node's parent, just in case that parent's height changed as well. In what scenario should we *not* continue to notify upward? Specifically, if after adding some value we see a node's **`.balance`** change (say, from 1) *to 0*, then that node's height did not change, and hence its parent chain is unaffected. Similarly, if after removing some value we see a node's **`.balance`** change from 0 to some other value, then that node's height did not change (try it with drawings on paper!). If a node's height didn't change, there's no need to keep checking parent nodes. This significantly optimizes our Update Balance Indices process.

Now that we know how to add and remove nodes from our AVL while updating the **`.balance`** appropriately, and now that we know how to rotate nodes to bring our tree back into balance, we are equipped to create the two most powerful methods on our AVLTree class.

Balanced Add

Using all we have learned in this chapter, create a **`balancedAdd(value)`** method for our AVLTree class. Ensure that by the time the method returns, our value is added, the tree is balanced, and all node attributes are updated and accurate.

Balanced Remove

Build **`balancedremove(value)`** for our AVLTree class. Ensure that when method returns (**`true`** if removed, **`false`** if not found), the value is removed, tree is balanced, and all node attributes are updated and accurate.

Rebalance

Similar to our Repair() function on a regular BST, create a Rebalance function for an AVLTree. Just as we could say that a Repair() function is not needed (since we expect the BST to insert and delete nodes correctly and hence never become invalid), similarly we could argue that Rebalance() is not needed since the AVL tree will continually keep itself balanced. Nonetheless, quickly build this using other methods you've already created.

Chapter 18 – Trees, Part III

There are other types of self-balancing tree as well. One example is the Red-Black Tree.

Red-Black Trees

A Red-Black Tree is based on our normal Binary Search Tree, plus these rules:

1. A Boolean within each node designates it as currently *red* or *black*.
2. The root node is black.
3. The `null` underneath each leaf node is considered black.
4. If a node is red, then both its children must be black.
5. Every path from node to descendent `null` contains the same number of black nodes.
   The uniform number of black nodes in paths from root to leaves is the tree's **black-height**.

As with the AVLTree, search methods in an RBTree (such as `contains`) are identical to those of a BST. The add and remove properties, however, are more interesting. To add a value to an RBTree, we create an RBNode (these default to Red) and insert it at the appropriate place in the tree. If the Red-Black rules are not violated, we are done. Otherwise, we either "repaint" certain nodes or we rebalance (and then repaint) certain nodes as necessary.

Red-Black Tree and Red-Black Node Class Definitions

Create the simplest possible class definitions of `RBTree` and `RBNode`.

Red-Black Add

Create the `add(value)` method on the `RBTree` class. As needed, repaint and/or rebalance; this is a self-balancing method.

Red-Black Remove

Create the RBTree's `remove(value)` method. As needed, repaint and/or rebalance nodes; this is a self-balancing method.

Short-Answer Questions on AVL and Red-Black Trees

- Self-balancing seem like a lot of work. When are costs *justified*? When are they *not justified*?
- Between AVL trees and Red-Black trees, which incurs more rebalancing cost?
- How would shapes of AVL trees and Red-Black trees generally differ (if at all)?
- What are the performance differences between these trees?
- When would you choose AVL tree over Red-Black, and vice-versa?
- Between AVL, Red-Black and BST, which `height()` is fastest? What is its big-O?

Chapter 18 – Trees, Part III

The final self-optimizing tree that we consider is one that does not actually automatically *balance* itself. Instead, it *optimizes* itself for how it is currently being used. This data structure, the **Splay Tree**, is flexible and adaptable to many usage scenarios.

Splay Trees

Splay Trees, like the AVL and Red-Black Trees, are based on generic Binary Search Trees, with additional rules. Here are the additional rules that enable this data structure to optimize itself:

1. When a value is added, the new node becomes the root of the tree;
2. On a search, the last node accessed (whether the *found* node or not!) becomes the root;
3. When a value is deleted, the *parent* of the removed node moves to become the root.
4. To promote nodes to root position, the tree uses a rotation operation called a Splay.

When a node is Splayed, we effectively perform a series of tree rotations until the node is moved to the root position. If the node is currently the left child of a left child, or if the node is currently the right child of a right child, then the 'grandparent' node is rotated twice. Otherwise the 'parent' node is rotated once, and then the 'grandparent' node once. This two-rotation cycle repeats as necessary. One last single rotation might be needed to move the node into the root position.

Splay Tree Class Definitions

Create the simplest possible class definitions of **`SplayTree`**. Do you need a **`SplayNode`**?

Splay Add

Create the **`add(value)`** method on the **`SplayTree`** class, splaying the new node.

Splay Contains

Create a **`contains(value)`** method for **`SplayTree`**, splaying as needed.

Splay Remove

Create a **`remove(value)`** method for **`SplayTree`**, splaying as needed.

Short-Answer Questions on Splay and Self-Balancing Trees

- How would shapes of Splay, AVL, Red-Black and BST trees generally differ (if at all)?
- What are the performance differences between Splay trees and BSTs?
- When would you choose Splay trees over AVL or Red-Black?
- Compare the likely performance of **`height()`** across Splay, Red-Black and BST.
- For the most recently accessed item in a Splay Tree, what is the big-O to remove it?

Chapter 18 – Trees, Part III

(TBD) B-Trees

Appendix – Brainteasers

These puzzles don't fit cleanly into any of the previous chapters, sometimes because they are not actually *coding* problems. However, they have been asked in technical interviews to assess candidates' reasoning abilities, and so we want to pose them to you here. With each of these, just as with the estimation challenges, think about the information and assets that you have, and try to make best use of those objects and information. Many companies are starting to move away from brainteaser puzzles like these, so don't worry if these sorts of questions are "not your thing". Just have fun with them!

(TBD) Logic & Numbers: Coins/Marbles and Scale

(TBD) Which Lockers are Open

(TBD) Light Switches and Bulbs

(TBD) Burning Fuses (1h each: 45min)

(TBD) Escaping the Train (2/3, @ 10mph)

(TBD) Cross Bridge (w/ flashlight) with 1, 2, 5, 10 in <19?

(TBD) Counting Cubes (3x3, 3x3x3, 4x4x4, NxNxN, NxNxNxN, N^i)

(TBD) Fox (4x) and Duck

(TBD) Fox and Rabbit)

Appendix – Graphics

(TBD) GFX: Draw Circle (1/8, given radius and setPixel(x,y))

(TBD) Rectangles Overlap?

(TBD) Draw skyline

Appendix – Concurrency

Multithreading and parallel computing
Map-Reduce
Hadoop

(TBD) Concurrency: Dining Philosophers

(TBD) Producer-Consumer

(TBD) Producers:Ever-Increasing Number