Algorithms Course Summary

CODING DOJO™

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 so you can get the handout and the introduction to the challenge.

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 working in teams of developers as a whole. Unless otherwise directed by the instructor, always work 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 it 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.

The "Basic 13"

CODING DOJO™

Print 1-255
Print all the integers from 1 to 255.

Odds 1-255
Print all odd integers from 1 to 255.

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

Iterate Array
Print all values in a given array.

Find Max
Print the largest element in a given array.

Get Average
Analyze an array's values and print the average.

Array with Odds
Create an array with odd integers from 1-255.

Square the Values
Given an array, square each value in the array.

Greater Than Y
Count the number of array values greater than a given value Y.

Zero Out Negative Numbers
Set negative array values to zero.

Max, Min, Average
Given an array, print max, min and average values.

Shift Array Values
Given an array, shift all values forward, dropping the first value and leaving an extra '0' value at the end.

Number to String
Replace any negative values in an array with 'Dojo'.

The "Basic 13" w/bugs (#1)

CODING DOJO™

Print1To255()

Print all the integers from 1 to 255.

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

PrintOdds1To255()

Print all odd integers from 1 to 255.

```
function printOdds1to255()
{
  var num = 1;
  while (num <= 255)
  {
    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++) {
    sum += num;
  }
  return sum;
}
```

PrintArrayVals(arr)

Print all values in a given array.

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

PrintMaxOfArray(arr)

Print the largest element in a given array.

```
function printArrayMax(arr)
{
  if (arr.length == 0) {
    console.log("Empty arr, 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);
}
```

PrintAverageOfArray(arr)

Analyze an array's values and print the average.

```
arr = [1,4,7,2,5,8];
if (arr.length == 0)
{
  console.log("Empty arr, 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 "Basic 13" w/bugs (#1)

CODING DOJO™

ReturnOddsArray1To255()

Create & return 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;
```

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];
  }
}
```

ReturnArrayCountGreaterThanY(arr, y)

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

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

ZeroOutArrayNegativeVals(arr)

Given an array, set negative values to zero.

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

PrintMaxMinAverageArrayVals(arr)

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

```
function maxMinAverage(arr)
{
  if (arr.length == 0) { return; }   CONSOLE.LOG("EMPTY ARRAY")
  var min = arr[0];
  var max = arr[0];          VAR AVG = 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];
  }                    SUM/ARR.LENGTH
  return min; -AVG =
  return max;
  return(avg/ARR.LENGTH);
}
```

CONSOLE.LOG

ShiftArrayValsLeft(arr)

Given an array, shift values forward by one, dropping the first value and leaving an extra '0' value at the end.

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

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;
}
```

2

ASSOCIATING ARRAY

```
VAR RETURNARR = {},
VAR ['MAX'] = MAX
VAR ['MIN'] = MIN
VAR ['AVG'] = AVG
```

RETURN RETURNARR

HTML PRE TAG

Week 1 - Fundamentals

CODING DOJO™

Overview
For this week, you will be familiarizing yourself with the basic constructs of programming: loops, conditionals, logic operators, and a few techniques.

Prerequisites
It is imperative that you complete the 13 mandatory algorithm challenges from earlier in the bootcamp.

Study Guide
Here is a list of concepts to study. Some or all of these will be used to solve this week'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)*

T-Diagrams
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 every algorithm challenge this week.

TDD , Tests, and Examples
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.

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

For Loops

```
for (INITIALIZATION; TEST; INCREMENT/DECREMENT)
{
    // body of the loop -- run 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 the 'if' statement -- only runs if CONDITION_1 is true AND CONDITION_2 is true
}
else
{
    // body of the 'else' statement -- only runs if CONDITION_1 is false or CONDITION_2 is false
}
```

Functions

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

// Calling standalone functions. ARGUMENTS passed by a caller enter the function as PARAMETERS
MY_FUNCTION(ARGUMENT_1, ARGUMENT_2);
```

Tomorrow: *sum fun*? Sigma and factorial

Week 1 - Fundamentals - '13' Review #1

CODING DOJO™

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 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;
  }
}
```

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);
  }
}
```

Iterate 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]);
  }
}
```

Find 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);
}
```

Get Average

Analyze an array's values and print the average.

```
function printArrayAverage(arr)
{
  if (arr.length == 0) {
    console.log("Null 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);
}
```

Tomorrow: *sum fun?* Sigma and factorial

Week 2 - Arrays 1 (of 2)

CODING DOJO™

Overview
This week, we explore the *array* data structure: reading and changing elements, as well as adding and removing elements (hence changing the array's length). Before week's end, we will also touch upon associative arrays as well.

Prerequisites
At this point we expect that you can quickly complete the earlier 13 mandatory algorithm challenges. Also, building straightforward Array functions such as min(), max(), sum() and average() should be easy and rapid.

T-Diagrams
Tracking variables with a t-diagram as you write an algorithm is extremely beneficial. Use a t-diagram for challenges this week.

Extra Goodies
Below are some of the Javascript constructs we'll use this week. Remember these basic building blocks!

Concepts
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).

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 new values are set beyond the current length.

Finally, values in a JavaScript array need not all be the same data type. A single array can contain numbers, booleans, strings, objects, functions, etc.

Declaring a new array:

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

Setting and accessing array values:

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

Array.length is determined by the largest index:

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

Overwriting array values:

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

Arrays can be sparsely populated:

```
myArray[myArray.length + 2] = 0.212;  //  myArray == [101, "hello!", true, undefined, 0.212]
console.log(myArray.length);          //  -> "5"
```

Shorten an array with the pop() method:

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

Week 2 - Arrays 1 - Monday

CODING DOJO™

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

for / while loops *array.pop() & push()* *can contain different data types*
if / else statements *arrays grow: arr.length == lastIdx-1* *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.

Answer:

```
FUNCTION PUSH FRONT (ARR){
 RETURN INSERTAT (ARR, 0, NUM)
}
```

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().

Answer:

```
FUNCTION POP FRONT (ARR){
 RETURN REMOVE A( ARR, 0)
}
```

InsertAt

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

Answer:

```
FUNCTION INSERT AT(ARR, INDEX, NUM){
  ARR.PUSH (ARR[ARR.LENGTH-1]);
  FOR (VAR i= ARR.LENGTH-2; i >INDEX; i--){
    ARR[i] = ARR[i-1];
  }
 ARR[INDEX] = NUM;
}
```

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().

Answer:

```
FUNCTION REMOVE AT (ARR, INDEX){
  VAR VALUE =ARR[INDEX];
  FOR( VAR i =INDEX; i <INDEX-1; i++){
   ARR[i] =ARR[i+1];
  }
 ARR POP( );
 RETURN VAUG;
}
```

Tomorrow: u-turns and censorship

Week 2 - Arrays 1 - Monday Recap

CODING DOJO™

PushFront

Insert a given value at the beginning of the given array, without using any built-in array methods.

```
function pushFront(arr, val)
{
  for (var idx = arr.length; idx > 0; idx--) {
    arr[idx] = arr[idx - 1];
  }
  arr[0] = val;
}               //  no "return arr" needed. Why?

// Arrs are objects passed by reference. A
// pointer (not a copy) of the arr is passed, so
// [] enables the method's code to cross the ptr
// and access the caller's real array

var myArray = [true, "schweet"];
pushFront(myArray, 42); // no return val needed
console.log(myArray);   // [42, true, "schweet"]
```

PopFront

Remove and return the first value of the given array, without using built-in array methods except pop().

```
function popFront(arr)
{
    if (arr.length === 0) {
        return null;
    }
    var returnVal = arr[0];

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

InsertAt

Insert a given value into given array at given index, without using built-in array methods. Similar to the above, with index instead of 0.

```
function insertAt(arr, index, val)
{
  if (index < 0)          { index = 0;  }
  if (index > arr.length) { index = arr.length;}
  for (var idx = arr.length; idx > index; idx--)
  {
    arr[idx] = arr[idx - 1];
  }
  arr[index] = val;
}


// ...and now that we have the above, we can do:
function pushFront2(arr, val) {
  insertAt(arr, 0, val);
}
```

RemoveAt

Remove and return the value in a given array and index, without using built-in array methods except pop(). Similar to the above, with index instead of 0.

```
function removeAt(arr, index)
{
  if (index < 0 || index >= arr.length) {
    return null;
  }
  var returnVal = arr[index];

  for (var idx = index + 1; idx < arr.length;
                            idx++) {
    arr[idx - 1] = arr[idx];
  }
  arr.pop();
  return returnVal;
}


// ...and now that we have the above, we can do:
function popFront2(arr) {
  return removeAt(arr, 0);
}
```

Week 2 - Arrays 1 - Tuesday

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

for / while loops *array.pop() & push()* *can contain different data types*
if / else statements *arrays grow: arr.length == lastIdx-1* *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.

Answer:

Remove Negatives

Implement a function removeNegatives() that accepts an array and removes any values that are less than zero. Optional: do this without two nested loops.

Answer:

Tomorrow: bread crumbs and ferris wheels

Week 2 - Arrays 1 - Tuesday Recap

CODING DOJO™

Reverse Array

Given an array, reverse the order of values. Reversed array should have same length, with elements moved to other indices so that order of elements is reversed.

```
//  Iterate arr end-->center, swapping values.
//  Be careful to handle both odd/even lengths.
function reverseArr(arr)
{
  for (var idx = arr.length - 1;
       idx >= arr.length / 2;
       idx--)
  {
    var temp = arr[idx];
    arr[idx] = arr[arr.length - idx - 1];
    arr[arr.length - idx - 1] = temp;
  }
}       // Voila - array has been reversed!
```

Remove Negatives

Implement removeNegatives() that accepts an array and removes values less than zero.

```
//  Iterate thru array, either counting/ignoring
//  a neg, or moving a non-neg forward to its
//  correct & rightful place. Then shorten array
//  by an appropriate amount (numNegsFound).
function removeNegs(arr)
{
  var numNegsFound = 0;
  for (var idx = 0; idx < arr.length; idx++) {
    if (arr[idx] >= 0) {
      arr[idx - numNegsFound] = arr[idx];
    } else {
      numNegsFound++;
    }
  }

  while (numNegsFound--) {
    arr.pop();
  }
//  or, even more briefly and simply:
  // arr.length -= numNegsFound;  //  !!

}       //  Viola!
```

Week 2 - Arrays 1 - Wednesday

CODING DOJO™

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

for / while loops
if / else statements
array.pop() & push()
arrays grow: arr.length == lastIdx-1
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.

Answer:

Rotate Array

Implement the function rotateArr(arr, shiftBy) that accepts an array and an offset. The function should shift the arr values by that amount. Values that shift off the array's end should 'wrap-around' to appear on the array's other side, so that no data is lost.

Optionally, add these advanced features:
a) negative shiftBy values (shift left instead of right),
b) minimize memory usage. With only a few local variables (not a partial array copy), the function should handle arrays and shiftBys that are in the millions,
c) minimize how many times you touch each element.

Answer:

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.

Answer:

Tomorrow: the languid largesse of lasts and largests

Week 2 - Arrays 1 - Wednesday Recap

CODING DOJO™

Binary Search

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

```
function binarySearchArr(arr, val)
{
  var startIdx = 0;
  var endIdx = arr.length;
  while (endIdx > startIdx) {
    var midIdx = Math.floor(
                 (startIdx + endIdx) / 2);
    if (val > arr[midIdx]) {
      startIdx = midIdx + 1;
    } else if (val < arr[midIdx]) {
      endIdx = midIdx;
    } else { return true; }
  }
  return false;
}
```

Min Of Sorted-Rotated

Find the min value in a sorted-then-rotated array.

```
function minIdxOfSortedRotatedArr(arr)
{
  var startIdx = 0;
  var endIdx = arr.length;
  while (endIdx - startIdx > 1) {
    var midIdx = Math.ceil(
                 (startIdx + endIdx) / 2);
    if (arr[startIdx] < arr[midIdx]) {
      startIdx = midIdx;
    } else { endIdx = midIdx; }
  }
  return startIdx;
}
```

Rotate Array

Implement rotateArr(arr, shiftBy) that moves arr values to the right by *shiftBy* spaces. Values that are shifted off the array's end should 'wrap-around' to appear at the array's beginning, so no data is lost.

Optionally, add these advanced features:
a) negative shifts (shift left instead of right),
b) minimize how much you touch each element,
c) minimize memory usage; a few local vars are OK, but create a solution that handles arrays as long as a million, with shiftBy of similar magnitude.

```
//  Handles negative and very large shiftBy
//  values, but is an inefficient O(n2)
//  solution. More about O(n2) later....

function rotateArr(arr, shiftBy)
{
  if (!arr.length || !shiftBy) { return; }

  while (shiftBy < 0) {
    shiftBy += arr.length;
  }
  shiftBy %= arr.length;

  while (shiftBy--) {
    var temp = arr[arr.length - 1];
    for (var idx = arr.length - 2;
         idx >= 0; idx--) {
      arr[idx+1] = arr[idx];
    }
    arr[0] = temp;
  }
}
```

Week 2 - Arrays 1 - Thursday

CODING DOJO™

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

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

Second-to-Last

Given an array, return the second-to-last element.

Answer:

Nth-to-Last

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

Answer:

Second-Largest

Given an array, return the second-largest element.

Answer:

Nth-Largest

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

Answer:

Tomorrow: the time-space continuum

Week 2 - Arrays 1 - Thursday Recap

CODING DOJO™

Second-to-Last

```
function secondLastVal(arr) {
  if (arr.length < 2) { return null; }
  return arr[arr.length - 2];
}
```

Nth-Largest

```
//  largest[] starts empty, build it as we
//  go. Compare arr[idx] to largest[] vals
//  and add it at the appropriate place.
function nthLargestVal(arr, n) {
  var largest = [];
  if ((arr.length < n) || (n < 1))
  { return null; }

  // For each val in given array...
  for (var idx = 0;idx < arr.length;idx++)
  {
  //...find where/if to insert in largest[]
    var largeIdx = largest.length - 1;
    while (  largeIdx >= 0
           && arr[idx] > largest[largeIdx])
    { largeIdx--; }

  //...shift largest[] values downward...
    var insertThis = arr[idx];
    for (var shiftIdx = largeIdx + 1;
         shiftIdx < largest.length;
         shiftIdx++) {
      var temp = insertThis;
      insertThis = largest[shiftIdx];
      largest[shiftIdx] = temp;
    }

  //...extend largest with last insertThis
    if (shiftIdx < n) {
      largest[shiftIdx] = insertThis;
    }
  }
  return largest[n-1];
}
```

Nth-to-Last

```
//  Extend secondLastVal to accept any 'n'
function nthLastVal(arr, n) {
  if ((arr.length < n) || (n < 1) ) {
    return null;
  }
  return arr[arr.length - n];
}
```

Second-Largest

```
function secondLargestVal(arr) {
  var largest = [];
  if (arr.length < 2) { return null; }
  largest[0] = Math.max(arr[0], arr[1]);
  largest[1] = Math.min(arr[0], arr[1]);
  for (var idx = 2; idx < arr.length;
                    idx++) {
    if (arr[idx] > largest[1]) {
      if (arr[idx] > largest[0]) {
        largest[1] = largest[0];
        largest[0] = arr[idx];
      }
      else { largest[1] = arr[idx]; }
    }
  }
  return largest[1];
}
```

Week 2 - Arrays 1 - Friday

CODING DOJO™

Concept: Time-space tradeoffs 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 week you will familiarize yourself with basic array manipulation. Here is a list of methods to study; some or all will be used in this week's challenges.

for / while loops *array.pop() & push()* *arrs are objects* *time-space tradeoff*
if / else statements *arrays grow: arr.length == lastIdx-1* *arrs contain different data types*

arrConcat

Replicate the array concat() function built into JavaScript. 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. Example: arrConcat(['a', 'b'], [1, 2]) should return ['a', 'b', 1, 2].

Answer:

Faster Factorial

Remember iFactorial from last week? 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.

Answer:

Next week: string theory

Week 2 - Arrays 1 - Friday Recap

CODING DOJO™

This week you became familiar with array basics, including these concepts:

for / while loops *array.pop() & push()* *arrs are objects* *time-space tradeoff*
if / else statements *arrays grow: arr.length == lastIdx-1* *arrs contain different data types*

arrConcat

Recreate the built-in function to concatenate two arrays. Do not change the two original arrays.

```
// Note: 'var arr3 = arr1' would not create
// a separate copy of the arr1 array; it
// would only put a copy of the arr1
// POINTER into the local variable arr3.
function arrConcat(arr1, arr2)
{
  var arr3 = [];
  for (var idx=0; idx<arr1.length; idx++) {
    arr3.push(arr1[idx]);
  }
  for (var idx=0; idx<arr2.length; idx++) {
    arr3.push(arr2[idx]);
  }
  return arr3;
}
```

Faster Factorial

Use time-space tradeoff to accelerate iFactorial(num).

```
//  Time/space: use space to accel runtime.
//  Store prev factorials & return them.
//  Otherwise, extend our array.
var factArr = [1];
function iBetterFactorial(num)
{
  if (num < 0)    {   return null;    }

  while (factArr.length <= num) {
    factArr.push( factArr[factArr.length-1]
                * factArr.length);
  }
  return factArr[num];
}
```

Week 2 - Arrays 1 - Weekend

CODING DOJO™

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

for / while loops
if / else statements
array.pop() & push()
arrays grow: arr.length == lastIdx-1
arrs are objects
arrs contain different data types
time-space tradeoff

For extra array practice this weekend, 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.

Answer:

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.

Answer:

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.

Answer:

Next week: linked list madness!

Week 3 - Strings 1 (of 2)

CODING DOJO™

Overview
This week, you will first briefly explore associative arrays, then continue onward into *strings* -- another special case of the basic indexed array.

Prerequisites
By now you should be very comfortable completing all 13 mandatory algorithm challenges from pre-boot-camp in less than two minutes each. Also, make sure to clearly understand the previous material.

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

Extra Goodies
At page bottom are examples of Javascript constructs we'll use this week. Remember these building blocks!

T-Diagrams
Being able to write a t-diagram to track your variables as you write an algorithm is extremely beneficial. Use a t-diagram for every algorithm challenge this week.

Concepts
Associative arrays (known as dictionaries, maps, or key-value pairs) are equivalent to objects in JavaScript. You can think of them as effectively arrays that can accept strings as indices. Empty associative arrays are initialized as { } instead of [ ], and they can be set or read by either object attribute syntax (array.attrib) or array index syntax (array["index"]).

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

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" }
```

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"
```

Challange: what is displayed by the following? Why?

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

Tomorrow: TLAs

Week 3 - Monday - Strings 1

CODING DOJO™

This week, you will first explore associative arrays, then continue into *strings*. Some or all of these methods will be used to solve this week's challenges.

.length *.split* *.join* *.concat* *for...in loops*

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], you should return {"abc": 42, 3: "wassup", "yo": true).

Answer:

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!

Answer:

InvertHash

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

Answer:

Challenge solved: `console.log(1 + 2 + "3" + "4" + 5 + 6)` displays the string "33456". num+num=num, but num+string=string, and string+num=string. 1+2=3, but 3+"3" = "33", and later "334"+5 = "3345".

Week 3 - Strings 1 - Monday Recap

CODING DOJO™

This week, you will first explore associative arrays, then continue into *strings*. Some or all of these methods will be used to solve this week's challenges.

.length .split .join .concat for...in loops

Arrs2Map

Given two arrays, create an associative array (map) containing keys of the first, and values of the second.

```
// Assoc array == map == hash == dictionary
// More keys than vals? Extra keys exist
// but == undefined. More vals than keys?
// Extra values are discarded
function arrs2Map(keyArr, valArr) {
  var returnMap = { };
  if (   keyArr instanceof Array
      && valArr instanceof Array) {
    for (var idx = 0; idx < keyArr.length;
             idx++) {
      returnMap[keyArr[idx]] = valArr[idx];
    }
  }
  return returnMap;
}
```

InvertHash

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

```
// (for...in) iterates a hash's keys. If
// keys share a val, val maps to LAST key
function invertHash(hash) {
  var newHash = {};
  if (typeof hash === "object") {
    for (key in hash) {
      var value = hash[key];
      newHash[value] = key;
    }
  }
  return newHash;
}
```

ReverseString

Implement reverseString(str) that will return the string of the same length but with characters reversed. Do not use the built-in reverse()!

```
// Copy str chars back->front to new arr;
// Join into string
function reverseStr(str)
{
  if (typeof str !== "string") {return "";}

  var arr = [];
  for (var idx = str.length - 1; idx >= 0;
                                 idx--) {
    arr.push(str[idx]);
  }
  return arr.join("");
}
```

Week 3 - Strings 1 - Tuesday Recap

CODING DOJO™

This week, we explore associative arrays, then continue into *strings*. For this week's challenges, some or all of these will be useful: *.length* *.split* *.join* *.concat* *for...in loops*

Remove Blanks

Given a string, return that string, with blanks removed.

```
//  Push non-blanks to arr; join into str
function removeBlanks(str) {
  if (typeof str !== "string"){return "";}

  var arr = [ ];
  for (var idx = 0; idx < str.length;
                    idx++) {
    if (str[idx] != " ") {
      arr.push(str[idx]);
    }
  }
  return arr.join("");
}
//     -- or even (if split(" ") is OK) --
function removeBlanks(str) {
  if (typeof str !== "string"){return "";}
  return str.split(" ").join("");
}
```

Get String Digits

Return a number made from the given string's digits.

```
// Go front->back looking for digits.
// For each, shift up & add.
function strDigits2Num(str)
{
  if (typeof str !== "string") { return 0;}
  var num = 0;
  for (var idx = 0; idx < str.length;
                    idx++) {
    if(str[idx] >= "0" && str[idx] <= "9"){
      num = num * 10 + parseInt(str[idx]);
    }
  }
  return num;
}
```

Acronyms

Given a string of words, return an acronym of the words in the original string.

```
// At start or post-space, next is initial.
// Join arr into string, make uppercase.
function acronym(str)
{
  if (typeof str !== "string") {return "";}
  var acronymArr = [ ];
  var nextIsInitial = true;
  for (var idx = 0; idx < str.length;
                    idx++) {
    if (str[idx] == " ") {
      nextIsInitial = true;
      continue;
    }
    if (nextIsInitial) {
      nextIsInitial = false;
      acronymArr.push(str[idx]);
    }
  }
  return acronymArr.join("").toUpperCase();
}
//     -- or even (if split(" ") is OK) --
function acronym(str)
{
  if (typeof str !== "string") {return "";}
  var acronymArr = [ ];
  var wordArr = str.split(" ");
  for (var idx = 0; idx < wordArr.length;
                    idx++) {
    if (wordArr[idx].length) {
      acronymArr.push(wordArr[idx][0]);
    }
  }
  return acronymArr.join("").toUpperCase();
}
```

Week 3 - Strings 1 - Wednesday

CODING DOJO™

Concept: 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 week we explored associative arrays then *strings*. Some or all of these will be useful in the challenges.

.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 an input of "**y(3(p)p(3)r)s**", return true. Given "**n(0(p)3**", return false. Given "**n)0(t(0)k**", return false.

Answer:

Braces Valid

Create a function that, given an input string, returns a boolean whether the various parentheses, braces and brackets in that string are valid. Given an input string of "**w(a{t}s[o(n{c}o)m]e)h[e{r}e]!**", return true. Given the input string "**d(i{a}l[t]o)n{e**", return false. Given the "**a(1)s[O(n]0{t}0}k**", return false.

Answer:

Tomorrow: Doc, note: able was I, ere I saw Adam.

Week 4 - Strings 2 - Tuesday

CODING DOJO™

This week, you will first explore associative arrays, then continue into *strings*. Some or all of these methods will be used to solve this week's challenges.

.length *.split* *.join* *.concat* *for...in loops*

Remove Blanks

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

Answer:

```
FUNCTION REMOVE BLANKS('STRING')
VAR NEWSTRING = STR.SPLIT("STRING")Answer:
```

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!", you should return "LFNYISN"

Get String Digits

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

Answer:

DEBUGGER FOR JAVASCRIPT

- BROWSER
- SOURCES

Tomorrow: parenthetically speaking....

Week 4 - Strings 1 - Wednesday Recap

This week's challenges focus on *strings*. Some or all of these might 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.

```
// Track nest level: always >= 0; end as 0
function parensValid(str)
{
  if (typeof str !== "string")
  { return false; }           // fast-fail

  var numParens = 0;
  for (var idx=0; idx<str.length; idx++) {
    if (str[idx] == "(") { numParens++;  }
    if (str[idx] == ")") { numParens--;  }
    if (numParens < 0)   { return false; }
  }
  return (numParens === 0);
}
```

Braces Valid

Return whether various parentheses, braces and brackets in the given string are valid.

```
// Any close must match most recent open!
function bracesValid(str)
{
  if (typeof str !== "string")
  { return false; }           // fast-fail

  var matches =
        { ")": "(", "}": "{", "]": "[" };
  var braceArr = [];
  for (var idx = 0;idx < str.length;idx++){
    switch (str[idx]) {
      case "(":
      case "{":
      case "[":
        braceArr.push(str[idx]);
        break;
      case ")":
      case "}":
      case "]":
        if (  braceArr[braceArr.length - 1]
           != matches[str[idx]] ) {
          return false;
        }
        braceArr.pop();
        break;
      default:
        continue;
    }
  }
  return (braceArr.length === 0);
}
```

Week 4 - Strings 1 - Thursday

CODING DOJO™

This week's challenges 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.

Answer:

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: given "My favorite racecar erupted!", return "e racecar e".

Answer:

Tomorrow: books that all end the same....

Week 4 - Strings 1 - Thursday Recap

CODING DOJO™

This week's challenges focus on *strings*. Some or all of these might be useful:

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

Is Palindrome

Create a function that, given an input string, returns a boolean whether the string is a palindrome (mirrored from beginning to end). Given "abba" or "racecar", return true. Given "Non", return false.

```
//  Compare from both sides toward center.
//  Fast-fail if inequal, else success.
function isPalindrome(str)
{
  if (typeof str !== "string")
  { return false; }            // fast-fail

  for (var idx = str.length - 1;
       idx >= str.length / 2; idx--)
  {
    if(str[idx] != str[str.length-1 - idx])
    { return false; }          // fast-fail

  }
  return true;
}
```

Longest Palindrome

Given an input string, return the longest palindrome substring. Given "what up, dada?", return "dad". Given "not much", return "n". Include spaces as well: given "My favorite racecar erupted!", return "e racecar e".

```
//
function longestPalin(str)
{
  if (typeof str !== "string")
  { return false; }            // fast-fail

  var longestStart = 0;
  var longestEnd = 0;

  for (var start = 0;
       longestEnd < str.length; start++) {
    for (var end = str.length - 1;
         end >= longestEnd; end--) {
      for (var runner = 0;
           runner < (end - start) / 2;
           runner++) {
        if (  str[start + runner]
           != str[end - runner]) {
          break; }
      }
      if (runner >= (end - start) / 2) {
        longestStart = start;
        longestEnd = end;
      }
    }
  }
  return str.slice(longestStart,
                   longestEnd + 1);
}
```

Week 4 - Strings 1 - Friday

CODING DOJO™

Concept: 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 week'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".

Answer:

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". When given ["nice", "ice", "sic"], return "".

Answer:

Week 4 - Strings 1 - Friday Recap

CODING DOJO™

This week's challenges focus on *strings*. Some or all of these might be useful:
.length *.split* *.join* *.concat* *for...in loops* *switch/case*

Book Index

Given a sorted array of page numbers, return a book index string. Combine consecutive pages into ranges.

```
// For each page, is it part of a range?
function bookIndex(pageArr)
{
  if (   !(pageArr instanceof Array)
      || !pageArr.length)
  { return ""; }              // fast-finish

  var indexStr = "";
  var rangeStarted = false;
  for (var arrIdx=1; arrIdx<pageArr.length;
       arrIdx++) {

    // Is page consecutive to prev one?
    if (pageArr[arrIdx] !==
        pageArr[arrIdx - 1] + 1) {
      // No, page not part of a range
      rangeStarted = false;
      indexStr += pageArr[arrIdx-1] + ", ";
    } else {
      // Yes. Create new range or extend?
      if (!rangeStarted) {
        // Create new range
        rangeStarted = true;
        indexStr +=pageArr[arrIdx-1] + "-";
      }
      // (If extending a range, do nothing)
    }
  }
  indexStr += pageArr[pageArr.length - 1];
  return indexStr;
}

//   Test Vectors:
// [10,20,21,30,31,32]=>"10, 20-21, 30-32"
// [10,11,20,21,22,30]=>"10-11, 20-22, 30"
// [10,11,12,20,30,31]=>"10-12, 20, 30-31"
```

Common Suffix

When given an array of words, returns the largest suffix (word-end) that is common between all words.

```
// Assume entire first word, then shorten
// as needed. Inner FOR loop (from word-end
// toward front) checks 'suffStart' thus is
// no longer than needed. Solution's final
// line assumes string.slice() is allowed.

function commonSuffix(wordArr)
{
  if (   !(wordArr instanceof Array)
      || !wordArr.length)
  { return ""; }              // fast-finish

  var suffStart = 0;
  var first = wordArr[0];
  var firstLen = first.length;

  for (var word = 1; word < wordArr.length;
           word++) {
    var next = wordArr[word];
    var wordLen = next.length;

    // Is next word shorter than suffix?
    if (wordLen < firstLen - suffStart)
    { suffStart = firstLen - wordLen; }

    // Note first & next lengths may differ
    for (var idx = 0;
         idx < firstLen-suffStart; idx++) {
      if (  first[firstLen - idx - 1]
          != next[ wordLen - idx - 1]) {
        suffStart = firstLen - idx;
        break;
      }
    }
  }
  return first.slice(suffStart);
}
```

Week 4 - Strings 1 - Weekend

CODING DOJO™

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

Concept: Why Algorithm Challenges Don't Allow You to Use Built-In Functions

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

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

***string*.concat(*string1, string2, ..., stringX*)** - Add string(s) to an existing one. Return the new string.

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

***string*.slice(*start,end*)** - Extract part of a string and return it in a new string. *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 a string into an 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; additional items should not be included in the array. Note: existing string is unaffected.

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

Week 4 - Lists 1 (of 2)

CODING DOJO™

Overview

This week we explore linked lists, a data structure used widely in lower layers such as backends, frameworks, runtimes or operating systems.

Prerequisites

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.

Concept

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

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

Not all machines are complex, just as not all objects are complicated. However, your code can add and remove aspects of objects on-the-fly, so this makes them different than, say, a boolean or a number, which always occupy the same amount of space in memory. Why does this matter? Well, if you have debugged your JavaScript code in the browser, then you understand the idea of a call stack. This is the series of function calls that led the computer to where it is right now. Whatever code is currently running, when the present function returns, the JavaScript runtime will look to that call stack to help it "remember" which function it came from, as well as the state of all the local variables in that calling function at the time when it called into another function. The runtime quickly stores local variables in the call stack as a part of changing the execution to another function. Setting aside space in the call stack for booleans and numbers is easy – regardless of value, numbers generally occupy a 64-bit chunk of memory. However, objects are more 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 allocations of this type. This memory is called the *heap*, and it is used for any memory needs that change over time or are not 100% predictable ahead of time. When you tell the system to look at your 'blueprint' and construct a 'machine' that corresponds to those plans, space for that object is set aside in the heap – enough space for all the attributes and functions for that object. When the object needs more space, it can expand into adjacent memory in the heap. During normal operation, the heap is wide-open for all kinds of large and small memory allocations. Whereas the call stack is like apartment space in a high-rise tower, the heap is more like Montana.

When you create an object and store it in a local `var`, the system doesn't put the object in that fixed memory slot the way it does for a number or a boolean. It creates space in the heap, and then into your local `var` it puts a *reference* to that memory. 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 – like email address or cell phone number. A *pointer* does exactly that – it *points* to where the object is found. If you have information to retrieve from (or store to) an object, you "go there" by simply referencing that pointer, followed by the attribute you want within the object. This could look like `myProject.name` or `myQuizzes[3]` or even calling `getAverage(myArr)`. Yes, arrays, strings and even functions are objects – referenced by . or [or (.

Week 4 - Lists 1 (of 2) - Day 1

CODING DOJO™

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

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

To the right 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.

Tomorrow: mins and maxes and means, oh my

Week 4 - Lists 1 - Monday Recap

CODING DOJO™

Our challenges use the listNode definition at left. Also, you will notice a terminology pattern in which adding a value to a list is referred to as *pushing* a value into the list. Likewise, removing a value from a list is referred to as *popping* a value off of the 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.

```
function pushFront(firstNode, value)
{
  var newHead = new listNode(value);
  newHead.next = firstNode;
  return newHead;
}
```

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.

```
function removeFront(firstNode)
{
  if (firstNode) {
    firstNode = firstNode.next;
  }
  return firstNode;
}
```

contains

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

```
function contains(list, value)
{
  while (list)
  {
// Note: 1 == true == [1] == "1",
// which is NOT really what we want.
// Instead use ===, to be very specific.
    if (list.val === value)
    {

// Leave immediately as soon as we find it!
      return true;
    }
    list = list.next;
  }

// If we got this far, we didn't find it!
  return false;
}
```

front

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

```
function front(list)
{
  if (!list) { return null; }
  return list.val;

//  - or following the one-liner -
  // return list ? list.val : null;
}
```

Week 4 - Lists 1 - Tuesday

CODING DOJO™

This week 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 week'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**.

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

length

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

min, max

Create function **min(node)**, returning a list's smallest value. Also create **max(node)**, returning largest.

average

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

display

Create **display(node)** for debugging that returns a string containing values in the linked list. Build what you wish **console.log(myList)** showed you!

Tomorrow: push 'em back, push 'em back, waaay back

Week 4 - Lists 1 - Tuesday Recap

CODING DOJO™

length

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

```
function length(list)
{
  var count = 0;
  while (list)
  {
    count++;
    list = list.next;
  }
  return count;
}
```

average

Create a function **average()** that returns (wait for it...) the *average* of the values contained in that list.

```
function average(list) {
  if (!list) return null;

  var sum = list.val;
  var count = 1;
  list = list.next;

  while (list) {
    sum += list.val;
    count++;
    list = list.next;
  }
  return sum / count;
}
```

display

Create **display(node)** for debugging that returns a string containing values in the linked list.

```
function display(list) {
  var displayStr = "List values: (";

  while (list) {
    if (typeof list.val === "string") {
      displayStr += "'" + list.val + "'";
    } else if (typeof list.val === "object"
          && list.val instanceof Array) {
      displayStr += "[" + list.val + "]";
    } else {
      displayStr += list.val;
    }

    list = list.next;
    if (list) {
      displayStr += ", ";
    }
  }

  displayStr += ")";
  return displayStr;
}
```

min, max

Create function **min(node)**, returning a list's smallest value. Also create **max(node)**, returning largest.

```
function min(list) {
  if (!list) { return null; }
  var lowVal = list.val;
  list = list.next;
  while (list) {
    if (list.val<lowVal) lowVal = list.val;
    list = list.next;
  }
  return lowVal;
}

function max(list) {
  if (!list) { return null; }
  var highVal = list.val;
  list = list.next;
  while (list) {
    if (list.val>highVal) highVal=list.val;
    list = list.next;
  }
  return highVal;
}
```

Week 4 - Lists 1 - Wednesday

CODING DOJO™

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

pushBack

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

popBack

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

Tomorrow: before and after - targeting specific values

Week 4 - Lists 1 - Wednesday Recap

CODING DOJO™

back

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

```
function back(list)
{
  if (!list) return null;

  while (list.next) {
    list = list.next;
  }
  return list.val;
}
```

popBack

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

```
function popBack(list)
{
  if (!list || !list.next) return null;

  var runner = list;
  while (runner.next.next) {
    runner = runner.next;
  }
  runner.next = null;
  return list;
}
```

pushBack

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

```
function pushBack(list, value)
{
  var newNode = new listNode(value);
  if (!list) return newNode;

  var runner = list;
  while (runner.next) {
    runner = runner.next;
  }
  runner.next = newNode;
  return list;
}
```

Week 4 - Lists 1 - Thursday

CODING DOJO™

This week we familiarize ourselves with basic manipulation of the *singly linked list* data structure. Here are some concepts used in this week'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;
}
```

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.

removeVal

Create `removeVal(list,value)` that removes the node with the given `value`. 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.

Tomorrow: splits and partitions -- breaking up is hard to do

Week 4 - Lists 1 - Thursday Recap

CODING DOJO™

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.

```
function appendVal(list, value, after)
{
  var newNode = new listNode(value);
  if (!list) return newNode;

  var runner = list;
  while (runner.next) {
    if (runner.val === after) { break; }
    runner = runner.next;
  }
  newNode.next = runner.next;
  runner.next = newNode;
  return 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.

```
function prependVal(list, value, before)
{
  var newNode = new listNode(value);
  if (!list || list.val === before) {
    newNode.next = list;
    return newNode;
  }

  var runner = list;
  while (runner.next) {
    if (runner.next.val===before) { break;}
    runner = runner.next;
  }
  newNode.next = runner.next;
  runner.next = newNode;
  return list;
}
```

removeVal

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

```
function removeVal(list, value)
{
  if (!list) return null;
  if (list.val === value) return list.next;

  var runner = list;
  while (runner.next) {
    if (runner.next.val === value) {
      runner.next = runner.next.next;
      break;
    }
    runner = runner.next;
  }
  return list;
}
```

Week 4 - Lists 1 - Friday

CODING DOJO™

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

Next week: if we remove duplicates from a list, does that make it DRY?

Week 4 - Lists 1 - Friday Recap

CODING DOJO™

splitOnVal

Create `splitOnVal(list,num)` that splits a list in two at node with value `num`, returning the list's latter half.

```
function splitOnVal(list, value) {
  var returnNode = null;
  if (list) {
    if (list.val === value) {
      returnNode = list;
      list = null;
    } else {
      var runner = list;
      while (runner.next) {
        if (runner.next.val === value) {
          returnNode = runner.next;
          runner.next = null;
        }
        else { runner = runner.next; }
      }
    }
  }
  return returnNode;
}
```

deleteGivenNode

Create a listNode method that removes itself from the list while keeping the list (pointers) valid. N.B. there must be one subsequent node, else **false** is returned.

```
this.removeSelf = function() {
  if (!node || !node.next) { return false;}
  node.val = node.next.val;
  node.next = node.next.next;
  return true;
}
this.removeSelf2 = function() {
  if (!node || !node.next) { return false;}
  var nextNode = node.next;
  for(var attr in node) {delete node[attr]}
  for(var attr in nextNode) {
    node[attr] = nextNode[attr];
  }
  return true;
}
```

partition

Create `partition(list,value)` that moves nodes with values *less than* value to earlier, and nodes with values *greater than* to be later than that node.

```
function partition(list, value) {
  var leftSideStart = null;
  var rightSideStart = null;
  var leftSideEnd, rightSideEnd;
  var match = null, runner = list;

  while (runner) {
    if (runner.val === value && !match)
    { match = runner; }
    else if (runner.val < value) {
      if (!leftSideStart)
      { leftSideStart = runner;    }
      else
      { leftSideEnd.next = runner; }
      leftSideEnd = runner;
    }
    else {
      if (!rightSideStart)
      { rightSideStart = runner;    }
      else
      { rightSideEnd.next = runner; }
      rightSideEnd = runner;
    }
    runner = runner.next;
  }

  list = rightSideStart;
  if (rightSideEnd)
  { rightSideEnd.next = null; }

  if (match) {
    match.next = list;
    list = match;
  }
  if (leftSideStart) {
    leftSideEnd.next = list;
    list = leftSideStart;
  }
  return list;
}
```

Next week: hall of mirrors (recursion)

Week 4 - Lists 1 - Weekend

CODING DOJO™

This week you familiarized yourself with basic manipulation of the *singly linked list* data structure. Here are concepts used in this week'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() { ... }
}
```

dedupe

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.

dedupeWithoutBuffer

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

Next week: hall of mirrors (recursion)

Week 5 - Recursion

CODING DOJO™

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

Prerequisites
It is imperative that you can confidently complete the 13 basic algorithms and previous weeks' challenges.

T-Diagrams
Writing a t-diagram to track a function's current state of a function is extremely beneficial, particularly when creating recursive algorithms. Use a t-diagram for at least one algorithm challenge each day this week.

Testing your algorithms
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 will respond when given unexpected inputs. Thinking of different possible "corner cases" ahead of time allows you to create much more resilient code that stands the test of time.

Concept
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? Well, one thing you could do is 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 have 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:

1) **Base cases:**
When a function *is* able to determine (and return) an answer immediately, this is called a *'base case'*. For example, if you successfully guessed my number, we know right away that the game is over. Conversely, if you look for *'spizzwink'* in a dictionary and find no word between *'spitz'* and *'splash'*, you know that *'spizzwink'* is not in that dictionary. Thus there are *positive* base cases as well as *negative* base cases.

2) **Forward progress:**
If a function cannot completely solve a problem, but it can narrow the range of possibilities, we call this *'forward progress'*. For example, learning that your 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 always 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.

3) **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. Thus, 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 my initial challenge, or a second request that came from itself! A recursive function behaves correctly either way, so it doesn't know or care about this distinction.

Week 5 - Recursion - Monday

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

Week 5 - Recursion - Monday Recap

CODING DOJO™

Concept: Memoization

No, it isn't a misspelling. Commonly, in dynamic programming technique we save previous results, referencing them in the future instead of recomputing them. This saved (*cached*) progress is called a *'memo'*. Sometimes 'memoization' is seen as an optional parameter, passed to a recursive function.

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.

```
// Base case: rSigma(0 or neg vals) = 0
// Otherwise: rSigma(n) = rSigma(n-1) + n

function rSigma(num)
{
  if (num < 1) { return 0; }

  return parseInt(num) + rSigma(num - 1);
}
```

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.

```
// Public version has no 'start' & 'end' --
// we add these memos in rBinSearch()
// We could have added start & end to the
// public version but this is a bit cleaner
// Each time we recurse, we move either the
// left or the right 'bookend' inward until
// either they meet or we find the val.
function arrayContains(arr, val)
{
  return rBinSearch(arr,val,0,arr.length);
}

// Base cases: a) found it, b) bookends met
// Else: fwd prog = moving our bookends in.
function rBinSearch(arr, val, start, end)
{
  if ( start >= end || val < arr[start]
                    || val > arr[end-1]) {
    return false;
  }                  //  didn't find it!

  var mid = parseInt((start + end) / 2);
  if (arr[mid] === val) {
    return true;
  }                  //  found it!

  if (arr[mid] > val) {
    end = mid;        // search start : mid
  } else {
    start = mid + 1;  // search mid+1 : end
  }
  return rBinSearch(arr, val, start, end);
}
```

Week 5 - Recursion - Tuesday

CODING DOJO™

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

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

Recursive Fibonacci

Write a function rFib(n) that, given a number *n*, recursively computes and returns the nth number in the Fibonacci sequence. As earlier, consider the first two (n = 0, n = 1) Fibonacci numbers to have values 0 and 1. Thus, rFib(2) = 0+1, or 1; rFib(3) = 1+1, or 2; rFib(4) = 1+2, or 3; rFib(5) = 2+3, or 5. Also, rFib(3.65) = rFib(3), or 2. Finally, rFib(-2) = rFib(0), or 0.

binaryStringExpansion

You will be given a string containing characters "0", "1", and "?". For every "?", either "0" or "1" characters are valid. Write a recursive function that returns an array of all valid strings that have "?" characters expanded into "0" or "1". Ex.: binStrExpand("1?01?1") should return ["100101", "100111", "110101", "110111"]. For this challenge, you can use string functions such as slice(), etc., but be frugal with their use, as they are expensive.

Week 5 - Recursion - Tuesday Recap

CODING DOJO™

rFibonacci

Write a function rFib(n) that, given a number *n*, recursively computes and returns the nth number in the Fibonacci sequence. As earlier, consider the first two (n = 0, n = 1) Fibonacci numbers to be 0 and 1. Thus, rFib(2) = 1; rFib(3) = 2; rFib(4) = 3; rFib(5) = 5.

```
function rFib(num)
{
  if (num < 1) { return 0; }
  if (num < 2) { return 1; }

  return rFib(num - 1) + rFib(num - 2);
}
```

binaryStringExpansion

You will be given a string containing characters "0", "1", and "?". For every "?", either "0" or "1" characters are valid substitutions. Write a recursive function that returns an array of all valid strings with "?" expanded into either "0" or "1". Ex.: binStrExpand("1?01?1") returns ["100101", "100111", "110101", "110111"]. You may use functions such as slice(), but be frugal with their use, as they are time-expensive.

```
function binStrExpansion(str, arr)
{
  if (arr === undefined) { arr = []; }

  var first = str.split("?",1)[0];
  var second = str.slice(first.length + 1);

  if (first.length == str.length) {
    arr.push(str);
  } else {
    binStrExpansion(first+"0"+second, arr);
    binStrExpansion(first+"1"+second, arr);
  }
  return arr;
}
```

Week 5 - Recursion - Wednesday

CODING DOJO™

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

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

Recursive "Tribonacci"

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

String In-Order Subsets

Create strSubsets(str). Given a string, return an array with all possible in-order character subsets. Given "abc", return ["", "a", "b", "c", "bc", "ac", "ab", "abc"].

Tomorrow: factorials and anagrams

Week 5 - Recursion - Wednesday Recap

CODING DOJO™

rTribonacci

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

```
// Cover negs with open-ended <; cover non-
// ints with "< myInt+1" not "<= myInt".

function rTrib(num)
{
  if (num < 2) { return 0; }
  if (num < 3) { return 1; }

  return   rTrib(num - 1)
         + rTrib(num - 2)
         + rTrib(num - 3);
}
```

stringSubsets

Given string, return an array of all *in-order* subsets of those characters. For example, stringSubsets("abc") = ["", "a", "b", "c", "bc", "ac", "ab", "abc"].

```
// Two memos: subStr (containing a partial
// string we've built so far), plus arr
// (containing valid setset strs). Whittle
// str down by 1 char each time; explore 2
// paths: 1) include char in our candidate
// subStr, and 2) don't. Once str is empty,
// add subStr to our list of valid subsets.

function allSubsets(str, subStr, arr)
{
  if (arr === undefined)    { arr = []; }
  if (subStr === undefined) { subStr = "";}

  if (str == "") {
    arr.push(subStr);
    return arr;
  }

  var first = str[0];   // consume str, one
  str = str.slice(1);   // char at a time.

  allSubsets(str, subStr, arr);
  allSubsets(str, subStr + first, arr);

  return arr;
}
```

Week 5 - Recursion - Thursday

CODING DOJO™

This week you will familiarize yourself with recursion. Some or all of the following important concepts will be used in this week'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) = 0. Examples: rFact(3) = 1 * 2 * 3 = 6. Also, rFact(6.5) = 1 * 2 * 3 * 4 * 5 * 6 = 720.

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().

Week 5 - Recursion - Thursday Recap

CODING DOJO™

rFactorial

Given a number, return its factorial (num!). If number is less than zero, treat as zero. If number is not an integer, treat as integer. For example, rFact(3) = 1 * 2 * 3 = 6; rFact(6.5) = 1 * 2 * 3 * 4 * 5 * 6 = 720. Note: rFact(0) = 1.

```
function rFact(num)
{
  if (num < 0)  { return 0; }
  if (num < 2)  { return 1; }

  return Math.floor(num) * rFact(num - 1);
}
```

stringAnagrams

Given a string, return an array containing all anagrams of that string (different sequences of that string's letters). Given "tar", return ["art", "atr", "rat", "rta", "tar", "tra"]. For this challenge, you may use built-ins such as split().

```
// strAnagrams()+rAnagrams() == 1 solution.
// Add each char in str to an anagram, then
// recurse, repeating until str == "".
// When we recurse, we want 1) remaining
// orig str, 2) the fragment of anagram
// we've built so far, and 3) an array to
// which we'll add completed anagrams.
// #1 is dynamic; #2 and #3 are memos.

function strAnagrams(str) {
  var arr = [];
  rAnagrams(str, "", arr);
  return arr;
}
```

```
function rAnagrams(str, subStr, arr) {
  if (str == "") { arr.push(subStr); }

  for (var idx = 0;idx < str.length;idx++)
  {
    var before = str.slice(0, idx);
    var oneChar = str[idx];
    var after = str.slice(idx + 1);
    rAnagrams(before + after,
              subStr + oneChar,
              arr);
  }
  return arr;
}
```

```
// This function is another, more consolidated solution

function strAnagrams2(str, subStr, arr) {
  arr    = arr    || [];              // If arr or subStr are undefined, define them.
  subStr = subStr || "";              // If str is empty, we've used all the chars, so
  if (str == "") { arr.push(subStr); }  // therefore subStr contains a complete anagram.

  for (var idx = 0; idx < str.length; idx++) {
    strAnagrams2(str.slice(0, idx) + str.slice(idx + 1), subStr + str[idx], arr);
  }
  return arr;
}
```

Week 5 - Recursion - Friday

CODING DOJO™

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

allValidNPairParens

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

Week 5 - Recursion - Friday Recap

CODING DOJO™

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.

```
function node(value)
{
  this.val = value;
  this.next = null;
  // ... + other attributes as needed
}

// Simple code, but so wasteful of call
// stack space that any list of significant
// length will result in stack overflow. In
// real world, iterative is preferred.
function rListLength(node)
{
  if (!node) { return 0; }

  return 1 + rListLength(node.next);
}
```

allValidNPairParens

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

```
// Could be consolidated into rValidParens
function allValidNParens(numParens)
{
  var arr = [];
  rValidNParens(numParens, 0, "", arr);
  return arr;
}

function rValidNParens(parens, opens,
                       subStr, arr)
{
  // No more parens & all are closed: DONE.
  if (!parens && !opens) {
    arr.push(subStr);
    return arr;
  }

  // One option is to close an open paren.
  if (opens) {
    rValidNParens(parens,       opens - 1,
                  subStr + ")", arr);
  }
  // Another option is to open a new paren.
  if (parens) {
    rValidNParens(parens - 1,   opens + 1,
                  subStr + "(", arr);
  }
  return arr;
}
```

Next week: Arrays (again)

Week 5 - Recursion - Weekend

CODING DOJO™

This weekend's 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 whether 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-level challenge: 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)`.

Week 6 - Arrays II

CODING DOJO™

Overview

This week covers additional challenges in *arrays*. As we work with this data structure, we grow our confidence and velocity in loops and conditionals. Now, if needed, we can use newer concepts like recursion.

Prerequisites

It is imperative that you can confidently and quickly complete the challenges from previous weeks. If this is not the case, take extra time after hours and during the weekend to review and strengthen your skills.

Interview Tips

Students sometimes wonder *what language* they should use in technical interviews. The best answer, of course, is "whatever language the interviewer tells you to use." What if you don't know ahead of time what language they prefer? Or what if you *do* know, but you are not strong in that language? Don't despair! The truth is that most interviewers will say "*Write in whatever language where you are most comfortable....*" Please keep in mind that they are also thinking "*...as long as it is an appropriate choice for this problem,*" so don't choose Ruby when interviewing to write graphics firmware, and don't choose Fortran for a front-end interview. In general, 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.

Week 6 - Arrays II - Monday

CODING DOJO™

This week we dive deeply into Arrays. Put yourself into the mindset of a technical interview during this week's algorithm challenges, 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?"*

Average

(Warmup) Return the average value of a given array.

Shuffle

Recreate the function built into JavaScript's array object. Efficiently shuffle a given array's values. Do you need to return anything from your function?

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. Ex.: [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**.

Tomorrow: game, set, match -- flat dupe boogie

Week 6 - Arrays II - Monday Recap

CODING DOJO™

Concept: Sets

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...).

Additionally, sometimes you want a set to maintain its values in a specific order (such as words in a dictionary). This type of collection is referred to as ***ordered sets***. Other sets do not require ordering (such as cell phone numbers in a group text message list): these are called ***unordered sets***. More on these tomorrow!

average

```
function average(arr) {
  if (   !(arr instanceof Array)
      || !arr.length) { return null; }
  var sum = 0;
  for (var idx = 0;idx < arr.length;idx++)
  { sum += arr[idx]; }
  return sum / idx;
}
```

isBalanced

Return whether array has intra-index balance point.

```
function isBalanced(arr) {
  if (!(arr instanceof Array))
  { return false; }
  var sum = 0;
  for(var idx = 0;idx < arr.length;idx++)
  { sum += arr[idx]; }

  var halfSum = 0;
  if (sum / 2 == halfSum)   { return true;}
  for(var idx = 0;idx < arr.length;idx++) {
    halfSum += arr[idx];
    if (sum / 2 == halfSum) { return true;}
  }
  return false;
}
```

shuffle

```
function shuffle(arr) {
  if (!(arr instanceof Array)) { return; }
  for(var idx=arr.length-1; idx>0; idx--) {
    // Pick randomly, swap it to the back.
    var randIdx =
         Math.floor(Math.random()*(idx+1));
    var temp = arr[idx];
    arr[idx] = arr[randIdx];
    arr[randIdx] = temp;
  }
}
```

balanceIndex

Return *index* w/ equal sums on either side (-1 if none).

```
function balanceIdx(arr) {
  if (!(arr instanceof Array)) {return -1;}
  var rightSum = 0;
  for(var idx = 0;idx < arr.length;idx++)
  { rightSum += arr[idx]; }

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

Week 6 - Arrays II - Tuesday

CODING DOJO™

During this week's 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 arrays and empty [ ] elements. Do not alter the array, but return a new array that retains the original order. Example: given [1, [2, 3], 4, [ ]], return a new array [1, 2, 3, 4].

Second-level challenge: Work *'in-place'* in the given array (cannot create another). Alter order if needed. Ex.: [1, [2, 3], 4, [ ]] you could change to [1, 3, 4, 2].

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

Remove Duplicates

Remove duplicate values 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-level challenge: Work *'in-place'* in given array. Alter order if needed (*stability* is not required). Ex.: [1, 2, 1, 3, 4, 2] you could change to [1, 2, 4, 3].

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

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

Tomorrow: merges and unions and intersections, oh my!

Week 6 - Arrays II - Tuesday Recap

CODING DOJO™

```
// Stably flatten an arr, return new array.
function flatten1(arr, newArr)
{
  if (!(arr instanceof Array))
  { return []; }
  if (newArr===undefined)
  { newArr=[]; }

  for(var idx = 0; idx < arr.length; idx++)
  {
    if (!(arr[idx] instanceof Array))
    { newArr.push(arr[idx]; }
    else if (arr[idx].length)
    { flatten1(arr[idx], newArr); }
  }
  return newArr;
}

// Flatten arr in-place (unstable is OK)
function flatten2(arr)
{
  if (!(arr instanceof Array)) { return; }

  var temp, idx = 0;
  while (idx < arr.length) {
    if (arr[idx] instanceof Array) {
      if (arr[idx].length == 0) {
        temp = arr.pop();
        if (idx == arr.length)
        { break; }
      }
      else {
        temp = arr[idx].pop();
        if (arr[idx].length)
        { arr.push(arr[idx]); }
      }
      arr[idx] = temp;
    }
    else { idx++; }
  }
}
```

```
// Stability easier with new arr
function dedupe1(arr)
{
  if (!(arr instanceof Array)) {return [];}

  var newArr = [];
  for (var idx = 0;idx < arr.length;idx++)
  {
    for(var runner=0; runner<idx; runner++)
    {
      if (arr[runner] === arr[idx])
      { break; }
    }
    if (runner == idx)
    { newArr.push(arr[idx]); }
  }
  return newArr;
}

// dedupe2 omitted --
// Actually easier to make it stable (3)

// Stable and in-place, but O(n2)
function dedupe3(arr)
{
  if (!(arr instanceof Array)) { return; }

  var numSkipped = 0;
  for(var idx = 0; idx < arr.length; idx++)
  {
    for(var runner=0; runner<idx; runner++)
    {
      if (arr[runner] === arr[idx])
      {
        numSkipped++;
        break;
      }
    }
    if (runner == idx)
    { arr[idx - numSkipped] = arr[idx]; }
  }
  arr.length -= numSkipped;
}
```

Week 6 - Arrays II - Tuesday Recap

CODING DOJO™

```
// If we find a nested array, use it
// as a target, like when we used newArr
// Uses a familiar helper function....
function flatten3(arr)
{
  if (!(arr instanceof Array)) { return; }

  var idx = 0;
  while (idx < arr.length)
  {
    if (arr[idx] instanceof Array)
    {
      flatHelper(arr, arr[idx], idx+1);
      var subLen = arr[idx].length;
      for (var sub=subLen-1; sub>=0; sub--)
      {
        arr[idx + sub] = arr[idx][sub];
      }
      arr.length = idx + subLen;
      idx = 0;
      continue;
    }
    idx++;
  }
}

// Eerily similar to flatten1 (:  This
// flattens rest of arr into a given target
function flatHelper(arr, target, startIdx)
{
  if (startIdx === undefined) {
    startIdx = 0;
  }
  for (var idx = startIdx;
       idx < arr.length; idx++)
  {
    if (!(arr[idx] instanceof Array)) {
      target.push(arr[idx]);
    }
    else if (arr[idx].length) {
      flatHelper(arr[idx], target);
    }
  }
  return target;
}
```

```
// Use an associative array (object) to
// cache when a value is first seen. O(n)
function dedupe4(arr)
{
  if (!(arr instanceof Array)) { return; }

  var numSkipped = 0;
  var values = {};
  for (var idx=0; idx<arr.length; idx++) {
    if (values[arr[idx]] !== undefined) {
      numSkipped++;
    }
    else {
      values[arr[idx]] = true;
      arr[idx - numSkipped] = arr[idx];
    }
  }
  arr.length -= numSkipped;
}
```

Tomorrow: if today was Felix, tomorrow is O

Week 6 - Arrays II - Wednesday

CODING DOJO™

Concept: Set Operations

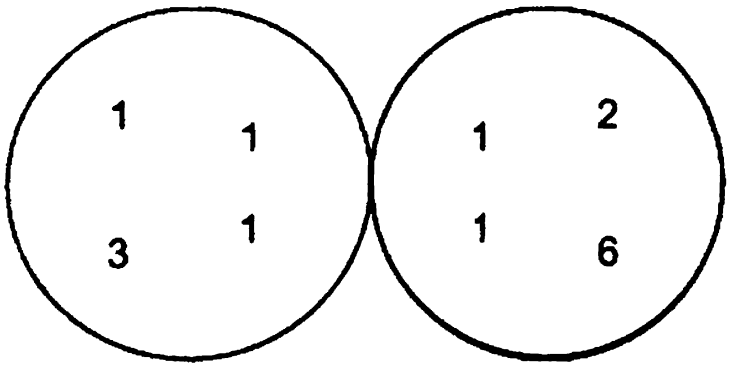
Yesterday we discussed different characteristics of a *set*. By default, sets contain no duplicates, but there is a type that can, called a *multiset*. A normal set does not keep track of the counts of values, but a multiset does.

A set that keeps its elements in strict order is called an *ordered set* (or *ordered multiset!*). One that doesn't do so is an *unordered set/multiset*. As we saw in yesterday's challenges, there are costs associated with removing duplicates, and with maintaining a set's order, so if we simply throw values into an array without sorting them or removing duplicates, we have an unordered multiset.

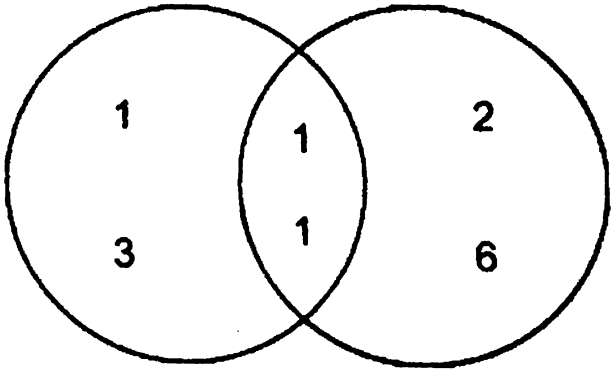
What can we do with these different flavors of set? When working with a set, you might *add* an element, *remove* one, get the quantity present, or check whether the set *contains* a certain element. With *ordered* sets, we can also use the ordering to retrieve *first* or *last* element, and from any 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.

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 two sets, together.



A *union* contains values in either set: (1, 3, 1, 1, 2, 6)



An *intersection* contains only values in both sets: (1, 1)

Finally, as shown above, for *multisets* these operations also affect *value counts* accordingly. For two multisets each containing the same value ('1' in the diagram above), the union will retain the higher of the counts (three '1's are retained); the intersection retains the lower count (two '1's are retained). Given ordered multisets [1, 1, 1, 3] and [1, 1, 2, 6], their *merge* is [1, 1, 1, 1, 1, 2, 3, 6], their *union* is [1, 1, 1, 2, 3, 6], and their *intersection* is [1, 1].

Tomorrow: if today was Felix, tomorrow is Oscar....

Week 6 - Arrays II - Wednesday

Throughout the week, remember set operations and characteristics:

merge *union* *intersection* *set/multiset* *ordered/unordered*

Merge Sorted Arrays

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

Union Sorted Arrays

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

Intersect Sorted Arrays

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

Tomorrow: if today was Felix, tomorrow is Oscar....

Week 6 - Arrays II - Wednesday Recap

CODING DOJO™

Combine Sorted Arrays

Merge two sorted arrays, keeping all elements.

```
function combine2Sorted(arr1, arr2) {
  if (!(arr1 instanceof Array)
   || !(arr2 instanceof Array)){return [];}
  var arr3 = [];
  var idx1 = 0, idx2 = 0;
  while (  idx1 < arr1.length
        && idx2 < arr2.length)
  {
    if (arr1[idx1] <= arr2[idx2])
    { arr3.push(arr1[idx1++]); }
    else
    { arr3.push(arr2[idx2++]); }
  }
  while (idx1 < arr1.length)
  { arr3.push(arr1[idx1++]); }
  while (idx2 < arr2.length)
  { arr3.push(arr2[idx2++]); }
  return arr3;
}
```

Union Sorted Arrays

Create the *union multiset* of two sorted arrays.

```
function union2Sorted(arr1, arr2) {
  if (!(arr1 instanceof Array)
   || !(arr2 instanceof Array)){return [];}
  var arr3 = [];
  var idx1 = 0, idx2 = 0;
  while (  idx1 < arr1.length
        || idx2 < arr2.length) {
    if (idx2 == arr2.length)
    { arr3.push(arr1[idx1++]); }
    else if (   idx1 == arr1.length
             || arr2[idx2] < arr1[idx1])
    { arr3.push(arr2[idx2++]); }
    else {
      if(arr1[idx1]===arr2[idx2]) {idx2++;}
      arr3.push(arr1[idx1++]);
    }
  }
  return arr3;
}
```

Intersect Sorted Arrays

Intersection multiset of two sorted arrays in new array.

```
function intersect2Sorted(arr1, arr2) {
  if (!(arr1 instanceof Array)
   || !(arr2 instanceof Array)){return [];}
  var idx1 = 0, idx2 = 0, arr3 = [];
  while (  idx1 < arr1.length
        && idx2 < arr2.length)
  {
    if (arr1[idx1] === arr2[idx2]) {
      arr3.push(arr1[idx1++]);
      idx2++;
    }
    else if (arr2[idx2] < arr1[idx1]) {
      idx2++;
    }
    else {
      idx1++;
    }
  }
  return arr3;
}
```

Intersect Sorted Arrays (in-place)

Get *intersection multiset* of sorted arrays, in-place.

```
function intersect2SortedInPlace(arr1,arr2)
{
  if (!(arr1 instanceof Array)
   || !(arr2 instanceof Array)){return [];}
  var numSkipped = 0, idx1 = 0, idx2 = 0;
  while (  idx1 < arr1.length
        && idx2 < arr2.length) {
    if (arr2[idx2] < arr1[idx1])
    { idx2++; }
    else {
      if (arr1[idx1] === arr2[idx2]) {
        arr1[idx1-numSkipped]=arr2[idx2++];
      }
      else { numSkipped++; }
      idx1++;
    }
  }
  arr1.length = idx1 - numSkipped;
}
```

Week 6 - Arrays II - Thursday

CODING DOJO™

This week, 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 the week, remember the basic set operations and characteristics:

union *intersection* *set / multiset* *ordered / unordered*

Union Sorted Arrays (dedupe)

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

Intersection Unsorted Arrays

Intersect two unsorted arrays, putting the *unsorted multiset* result 'in-place' into the first. Given [2,7,2,1] and [6,7,2,7,6,2], you could change the first to [7,2,2].

Second challenge: don't alter the arrays; return a new one. Is this easier? Faster?

Union Unsorted Arrays

Given two unsorted arrays, return a new unsorted one with the the *union multiset*; do not alter the two original arrays. Given [2, 7, 2, 1] and [6, 7, 2, 7, 6, 2], you could return [7, 2, 7, 2, 1, 6, 6]. How efficient can you make this, in case of *long* arrays?

Week 6 - Arrays II - Thursday Recap

CODING DOJO™

Union Sorted Arrays (no duplicates)

Create the *union set* of two sorted arrays. Set (not multiset) means you should remove duplicates.

```
function unionSortedNoDupe(arr1, arr2)
{
  if (!(arr1 instanceof Array) || !(arr2 instanceof Array))    { return []; }
  var idx1 = 0, idx2 = 0, arr3 = [];
  while (idx1 < arr1.length || idx2 < arr2.length) {
    while (idx1 < arr1.length && arr1[idx1+1] === arr1[idx1])  { idx1++; }
    while (idx2 < arr2.length && arr2[idx2+1] === arr2[idx2])  { idx2++; }
    if (idx2 == arr2.length)                                   { arr3.push(arr1[idx1++]); }
    else if (idx1 == arr1.length || arr2[idx2] < arr1[idx1])   { arr3.push(arr2[idx2++]); }
    else {
      if (arr1[idx1] === arr2[idx2])                           { idx2++; }
      arr3.push(arr1[idx1++]);
    }
  }
  return arr3;
}
```

Intersection Unsorted Arrays

Put *intersection multiset* of 2 arrays 'in-place' into first.

```
function intersectUnsorted(arr1, arr2)
{
  if (!(arr1 instanceof Array)
   || !(arr2 instanceof Array)) { return; }
  var counts = {};
  for(var idx2=0;idx2<arr2.length;idx2++) {
    if (counts[arr2[idx2]] === undefined)
    { counts[arr2[idx2]] = 0; }
    counts[arr2[idx2]]++;
  }
  var numSkipped = 0;
  for(var idx1=0;idx1<arr1.length;idx1++) {
    if (  counts[arr1[idx1]] !== undefined
       && (--counts[arr1[idx1]]) >= 0) {
      arr1[idx1 - numSkipped] = arr1[idx1];
    }
    else { numSkipped++; }
  }
  arr1.length -= numSkipped;
}
```

Union Unsorted Arrays

Return the *union multiset* of two unsorted arrays.

```
function unionUnsorted(arr1, arr2) {
  if (!(arr1 instanceof Array)
   || !(arr2 instanceof Array)){return [];}

  var arr3 = [];
  var counts = {};
  for(var idx1=0;idx1<arr1.length;idx1++) {
    if (counts[arr1[idx1]] === undefined)
    { counts[arr1[idx1]] = 0; }
    counts[arr1[idx1]]++;
    arr3.push(arr1[idx1]);
  }
  for(var idx2=0;idx2<arr2.length;idx2++) {
    if (   counts[arr2[idx2]] === undefined
        || (--counts[arr2[idx2]]) < 0) {
      arr3.push(arr2[idx2]);
    }
  }
  return arr3;
}
```

Week 6 - Arrays II - Friday

CODING DOJO™

This week we dive deeply into Arrays. Put yourself into the mindset of a technical interview during this week'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?"* *merger* *union* *intersection* *set / multiset* *ordered / unordered* *in-place* *stable*

If needed, refer to the solution to the "Union Unsorted Arrays" challenge for starting points to these:

Union Unsorted Arrays (in-place)

Put *union multiset* of 2 unsorted arrays into first. Given [2,7,2,1] and [6,7,2,6], change the first to [2,7,2,1,6,6].

Union Unsorted Arrays (no duplicates)

Return the *union set* (remove duplicates) of unsorted arrays. Given [2,7,2,1] and [6,7,2,6], return [2,7,1,6].

Next week: prepare to be re-enlisted

Week 6 - Arrays II - Friday Recap

CODING DOJO™

Union Unsorted (in-place)

Create the *union multiset* of two unsorted arrays, 'in-place' in the first array. Example: given [2,3,1] and [6,3,2,6], change the first array to [2,3,1,6,6].

```
// Similar to the previous, but in-place.
// Keep arr1 entirely, and push arr2 values
// into arr1 if they are not found in arr1,
// or if there are more of that value in
// arr2 than in arr1.
function union2UnsortedInPlace(arr1, arr2)
{
  if (!(arr1 instanceof Array)
   || !(arr2 instanceof Array)) { return; }

  var counts = {};

  for(var idx1=0; idx1<arr1.length; idx1++)
  {
    if (counts[arr1[idx1]] === undefined) {
      counts[arr1[idx1]] = 0;
    }
    counts[arr1[idx1]]++;
  }
  for(var idx2=0; idx2<arr2.length; idx2++)
  {
    if (   counts[arr2[idx2]] === undefined
       || (--counts[arr2[idx2]]) < 0) {
      arr1.push(arr2[idx2]);
    }
  }
}
```

Union Unsorted (no dupes)

Given two unsorted arrays, combine into a new sorted array containing the *union set* (i.e. remove duplicates). Example: given [2,3,1] and [6,3,2,6], return [2,3,1,6].

```
// Similar to yesterday's union challenge,
// in that we cannot alter the two original
// arrays. Do not allow duplicates into the
// union array. For this reason, 'counts'
// is now 'values' that holds only bools.
function union2UnsortedNoDupes(arr1, arr2)
{
  if (!(arr1 instanceof Array)
   || !(arr2 instanceof Array)){return [];}

  var values = {};
  var arr3 = [];
  for(var idx1=0; idx1<arr1.length; idx1++)
  {
    if (values[arr1[idx1]] === undefined) {
      values[arr1[idx1]] = true;
      arr3.push(arr1[idx1]);
    }
  }
  for(var idx2=0; idx2<arr2.length; idx2++)
  {
    if (values[arr2[idx2]] === undefined) {
      values[arr2[idx2]] = true;
      arr3.push(arr2[idx2]);
    }
  }
  return arr3;
}
```

Below is a weekend challenge. How efficient can you be? The input array might contain many millions of values.

Max Subarray Sum

Given an array, return the maximum sum of values from a sub-array. Any *consecutive sequence* of indices in the array is considered a sub-array. Create a function that returns the highest sum possible from these subarrays. Given [1,2,-4,3,-2,3], you should return **4** (for [3,-2,3]), and given [-1,-2,-4,-3,-2,-3], return **0** (for empty: [ ]).

Next week: prepare to be re-enlisted....

Week 7 - Strings II

CODING DOJO™

Overview

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

Recall that strings are *immutable*. This means that individual elements (characters) within the set cannot be altered, but the entire string object can be replaced by a new one. For this reason, it works (even if it feels like cheating) to 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 the algorithm challenges this week, you cannot use any built-in string methods unless they are explicitly mentioned. This means that you must work with strings simply as immutable arrays of characters. One exception: when capitalization is mentioned, then you are allowed to use the .toLowerCase() and .toUpperCase() methods.

Week 7 - Strings II - Monday

CODING DOJO™

String2WordArray

Create a function that, given a string of words (with spaces, tabs and linefeeds), returns an array of words. Example: 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. "This is a test", return "test a is This".
Bonus: also 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 of words, return a string with *only words that occur once* in that string. Ex.: given "Sing! Sing a song; sing out loud; sing out strong.", return "Sing! Sing a song; loud; strong". Note: treat punctuation as part of the word: "Sing!" is different than "Sing".
Bonus: ignore punctuation and capitalization. Ex.: given "Sing song! Sing a song; sing out loud and strong.", return "a loud and strong".

Week 7 - Strings II - Monday recap

CODING DOJO™

String2WordArray

```
function str2WordArr(str) {
  if (typeof str !== "string") {return [];}
  var wordArr = [], word = "";

  for (var idx = 0; idx<str.length; idx++){
    if (str[idx] == " " || str[idx] == "\t"
     || str[idx] == "\n") {
      if (word.length) {
        wordArr.push(word);
        word = "";
      }
    }
    else { word += str[idx]; }
  }
  if (word.length) { wordArr.push(word); }
  return wordArr;
}
```

Longest Word

```
function longestWord(str)
{
  if (typeof str !== "string")
  { return ""; }

  var wordArr = str2WordArr(str);
  var longWord = "";
  for (var idx=0;idx<wordArr.length;idx++)
  {
    if ( wordArr[idx].length
       > longWord.length)
    { longWord = wordArr[idx]; }
  }
  return longWord;
}
```

Reverse Word Order

Reverse the sequence of words in a string.

```
function reverseWordOrder(str)
{
  if (typeof str !== "string")
  { return ""; }

  var words = str2WordArr(str);

  for (var idx=0;idx <= (words.length/2)-1;
       idx++) {
    var temp = words[idx];
    words[idx] = words[words.length-idx-1];
    words[words.length - idx - 1] = temp;
  }

  return words.join(" ");
}
```

Unique Words

Discard all words except those that appear only once.

```
function uniqueWords(str) {
  if (typeof str !== "string") {return "";}

  var idx, uniqWords = "", words = {};
  var wordArr = str2WordArr(str);

  for (idx = 0;idx < wordArr.length;idx++){
    if (words[wordArr[idx]] === undefined)
    { words[wordArr[idx]] = 0; }
    words[wordArr[idx]]++;
  }

  for (idx = 0;idx < wordArr.length;idx++){
    if (words[wordArr[idx]] == 1) {
      if (uniqWords.length > 0)
      {uniqWords+=" ";}
      uniqWords += wordArr[idx];
    }
  }
  return uniqWords;
}
```

Week 7 - Strings II - Tuesday

CODING DOJO™

Overview

This week 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.

rotateString

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 the string "Boris Godunov" and value 5, return "dunovBoris Go".

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!" and ["crack", "poop"], return the string "Snap xxxxxle pop nincomxxxx!".
Bonus: handle capitalization appropriately.

isRotation

Create a standalone function that, given two strings, returns whether the second is a rotation of the first. Would you change your implementation if you knew that usually the strings would be entirely unrelated?

Week 7 - Strings II - Tuesday recap

CODING DOJO™

rotateString

Rotate string right. "Dessert" and 3 returns "ertDess".

```
function rotateStr(str, rotBy) {
  if (typeof str !== "string"){return "";}
  if (!str.length)         { return str; }
  if (!rotBy % str.length) { return str; }

  while (rotBy > 0) {rotBy -= str.length;}
  var idx, newStr = "";

  for (idx = str.length;
       idx < str.length * 2; idx++) {
    var shiftIdx = (idx-rotBy)%str.length;
    newStr += str[shiftIdx];
  }
  return newStr;
}
```

isRotation

Return whether one string is a rotation of another.
Bonus: handle capitalization.

```
function isRotation(str1,str2,ignoreCase) {
  if (   typeof str1 !== "string"
      || typeof str2 !== "string")
  { return false; }

  var len = str1.length;
  if (len != str2.length) { return false; }
  if (ignoreCase === true)   {
    str1 = str1.toLowerCase();
    str2 = str2.toLowerCase();
  }

  for (var rotate=0; rotate<len; rotate++){
    for (var idx = 0; idx < len; idx++) {
      var idx2 = (idx + rotate) % len;
      if (str1[idx] != str2[idx2]) {break;}
    }
    if (idx == len) { return true; }
  }
  return false;
}
```

Censor

Change naughty words to 'x's. **Bonus:** capitalize.

```
function censor(str, wordArr) {
  if (   typeof str !== "string"
      || !(wordArr instanceof Array))
  { return ""; }
  // Go through each naughty word...
  for (var arrIdx=0; arrIdx<wordArr.length;
       arrIdx++) {
    var word = wordArr[arrIdx];
    // ...go through the given string...
    for(var sIdx=0;sIdx<str.length;sIdx++){
      // ...match char-for-char with word
      for (var wIdx = 0;wIdx < word.length;
           wIdx++) {
        if (sIdx+wIdx>=str.length) {break;}
        var wordChar = word[wIdx];
        var strChar = str[sIdx + wIdx];
        // ignore case, break on mismatch.
        if ( wordChar.toLowerCase()
          !== strChar.toLowerCase())
        { break; }
      }
      // If entire word matched...
      if (wIdx == word.length) {
        var newStr = "";
        for (var newIdx = 0;
             newIdx < str.length;newIdx++) {
          if (  newIdx <  sIdx
             || newIdx >= sIdx + wIdx)
          { newStr += str[newIdx]; }
          else { //...then censor that part
            if(str[newIdx]>"Z")
            { newStr += "x"; }
            else { newStr += "X"; }
          }
        }
        str = newStr;
        sIdx += word.length;
      }
    }
  }
  return str;
}
```

Tomorrow: my one and only uniqueness

Week 7 - Strings II - Wednesday

Overview

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

isPermutation

Create a function that returns whether the second given string is a permutation of the first. For example, given "mister" and "stirner", return true. Given "mister" and "sister", return false.
Bonus: handle uppercase/lowercase.

isPangram

Create a function that returns whether a given string contains all letters in the English alphabet (upper or lower case). For example, given "How quickly daft jumping zebras vex!", return true. Given "abc def ghi jkl mno pqrs tuv wxy, not so fast!", return false.

allPermutations

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?

isPerfectPangram

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.

Week 7 - Strings II - Wednesday recap

CODING DOJO™

Overview

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

isPermutation

Bonus: handle uppercase/lowercase.

```
function isPermutation(str1, str2, ignoreCase){
  if ( typeof str1 !== "string"
      || typeof str2 !== "string")
  { return false; }
  if (str1.length != str2.length){return false;}
  if (ignoreCase === true) {
    str1 = str1.toLowerCase();
    str2 = str2.toLowerCase();
  }
  var idx, ltrs = {};
  for (idx = 0; idx < str1.length; idx++) {
    if (ltrs[str1[idx]] === undefined)
    { ltrs[str1[idx]] = 0; }
    if (ltrs[str2[idx]] === undefined)
    { ltrs[str2[idx]] = 0; }
    ltrs[str1[idx]]++;
    ltrs[str2[idx]]--;
  }
  for (ltr in ltrs) {
    if (ltrs[ltr] != 0) { return false; }
  }
  return true;
}
```

allPermutations

Did you cover all cases?

```
function allPerms(str, fragment, results) {
  if (typeof str !== "string") { return [];   }
  if (results === undefined)   { results=[];  }
  if (fragment === undefined)  { fragment=""; }

  if (str === undefined || !str.length) {
    if(fragment.length){results.push(fragment);}
    return results;
  }
  for (var idx = 0; idx < str.length; idx++) {
    var newStr = str.slice(0, idx);
    newStr += str.slice(idx + 1, str.length);
    allPerms(newStr,fragment+str[idx],results);
  }
  return results;
}
```

isPangram

Return whether string contains all English letters (u/l).

```
function isPangram(str) {
  if (typeof str !== "string") { return false; }
  str = str.toLowerCase();
  var ch, ltrs = [];
  for (idx = 0; idx < str.length; idx++) {
    ch= str.charCodeAt(idx) - "a".charCodeAt(0);
    if (  ch >= 0 && ch <= 25) {
      ltrs[ch] = true;
    }
  }
  for (ch = 0; ch < 26; ch++) {
    if (ltrs[ch] !== true) { return false;}
  }
  return true;
}
```

isPerfectPangram

Return whether string contains all English letters *once and only once*, ignoring punctuation and spaces.

```
// Lower the string. For each char, fail if
// prev found. Fail if all 26 ltr not found
function isPerfectPangram(str) {
  if (typeof str !== "string") { return false; }
  str = str.toLowerCase();
  var ch, ltrs = [];
  for (idx = 0; idx < str.length; idx++) {
    ch= str.charCodeAt(idx) - "a".charCodeAt(0);
    if (ch >= 0 && ch < 26) {
      if (ltrs[ch] === true) { return false;}
      ltrs[ch] = true;
    }
  }
  for (ch = 0; ch < 26; ch++) {
    if (ltrs[ch] !== true)   { return false;}
  }
  return true;
}
```

Week 7 - Strings II - Thursday

CODING DOJO™

Overview

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

dedupeString

Create a function that removes duplicate characters from a string (case-sensitive). Keep only the *last* instance of each character, including punctuation. Given "Snaps! crackles! pops!", return "Snrackle ops!".

idxFirstUniqLtr

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] == "k").

uniqueLetters

Create a function that returns 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".

Week 7 - Strings II - Thursday recap

CODING DOJO™

Overview

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

dedupeString

Keep only *last* instance of each character (case-sens).

```
function dedupeString(str, ignoreCase) {
  if (typeof str !== "string") {return "";}
  var input = str, deduped = "", ltrs = {};
  if (ignoreCase === true)
  { str = str.toLowerCase(); }

  for (idx = 0; idx < str.length; idx++) {
    if (ltrs[str[idx]] === undefined)
    { ltrs[str[idx]] = 0; }
    ltrs[str[idx]]++;
  }
  for(idx = str.length - 1;idx >= 0;idx--){
    if (ltrs[str[idx]] >= 1) {
      deduped = input[idx] + deduped;
      ltrs[str[idx]] = 0;
    }
  }
  return deduped;
}
```

idxFirstUniqLtr

Return index of first unique (case-sensitive) character.

```
function idxFirstUniqLtr(str, ignoreCase) {
  if (typeof str !== "string") {return -1;}
  if (ignoreCase) { str=str.toLowerCase();}

  var ltrs = {}, firstIdx = str.length;
  for (idx = 0; idx < str.length; idx++) {
    if (ltrs[str[idx]] === undefined)
    { ltrs[str[idx]] = idx; }
    else { ltrs[str[idx]] = str.length; }
  }
  for (ltr in ltrs) {
    firstIdx=Math.min(firstIdx, ltrs[ltr]);
  }
  if (firstIdx == str.length)  {return -1;}
  return firstIdx;
}
```

uniqueLetters

Return only the unique characters (case-sensitive).

```
function uniqueLetters(str, ignoreCase) {
  if (typeof str !== "string")
  { return ""; }

  var input = str;
  if (ignoreCase === true)
  { str = str.toLowerCase(); }

  var unique = "", ltrs = {};
  for (idx = 0; idx < str.length; idx++) {
    if (ltrs[str[idx]] === undefined)
    { ltrs[str[idx]] = 0; }
    ltrs[str[idx]]++;
  }
  for (idx = 0; idx < str.length; idx++) {
    if (ltrs[str[idx]] == 1)
    { unique += input[idx]; }
  }
  return unique;
}
```

This weekend: pushing on a string (to make it smaller)

Week 7 - Strings II - Friday

Overview

This week we revisit strings, now that we have mastered the fine art of *recursion*. Remember that recursion is not an answer to every problem; in fact 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 challenge: include fractional values as well. Given 11.2051, return the string "11.2051".

num2Text

Create a function that converts integer into string with the English text representation of the number. Given 40213, return "forty thousand two hundred thirteen".

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

Week 7 - Strings II - Friday recap

CODING DOJO™

num2String - **Bonus:** include fractional values: given 11.2051, return "11.2051".

```
function num2Str(num) { return "" + num; }   // That's it !!
```

num2Text - **Bonus challenge:** include 4 fractional digits. Given 11.2051, return "eleven point two zero five one".

```
var digitStr=["", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten", "eleven",
"twelve", "thirteen", "fourteen", "fifteen", "sixteen", "seventeen", "eighteen", "nineteen"];

var tensStr = ["", "", "twenty", "thirty", "forty", "fifty", "sixty", "seventy", "eighty", "ninty"];
var baseStr=["", " thousand", " million", " billion", " trillion", " quadrillion", " quintillion"];

function hundreds2Text(num) {
  if (num < 20)     { return digitStr[num]; }
  var returnStr = "";
  var ones = num % 10;
  var tens = Math.floor(num / 10);
  returnStr = tensStr[tens];
  if (ones && tens) { returnStr += "-"; }
  returnStr += digitStr[ones]
  return returnStr;
}

function thousands2Text(num) {
  var returnStr = "";
  var hundreds = Math.floor(num / 100);
  var low2Digits = num % 100;

  if (num >= 100) {
    returnStr = digitStr[hundreds] + " hundred";
    if (low2Digits) { returnStr += " "; }
  }
  returnStr += hundreds2Text(low2Digits);
  return returnStr;
}

function num2Text(num) {
  if (!num)   { return "zero"; }
  var base = 0, returnStr = "";

  while (num) {
    var low3Digits = num - (Math.floor(num/1000) * 1000);
    if (low3Digits)
    { returnStr = thousands2Text(low3Digits) + baseStr[base] + returnStr; }
    base++;
    num = Math.floor(num/1000);
    if (num && low3Digits) { returnStr = " " + returnStr; }
  }
  return returnStr;
}
```

Week 7 - Strings II - Weekend

CODING DOJO™

Overview

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

stringEncode

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 false.

Shortener

Given string and desired length, return the maximally readable string of that length. Suggested sequence: 1) capitalize each word, 2) remove spaces, starting from the back, 3) remove punctuation, from the back, 4) remove lower-case letters (vowels first), from the back, 5) remove upper-case letters, from the back.
Examples, given "It's a wonderful life!" as the string:
25: "It's a wonderful life! "
20: "It's AWonderfulLife!"
15: "ItsAWonderfulLf"
10: "ItsAWndrfL"
5: "ItAWL"

stringDecode

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

Next week: assorted sorts

Week 8 - Linked Lists II

CODING DOJO™

This week we visit an old friend, the *linked list*. This data structure, and *object pointer manipulation* in general, is so important that it demands more time. Before week's end we will see a new list type where each node connects to *two* others, but in our original linked list data structure, each node links only to a single node: the *next* one in sequence. This is why it is most accurately called a *singly linked list*. This makes it easy to iterate *forward*, going backward is more tricky. For challenges this week, use the following 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.contains = function(value) {
    var runner = this.head;
    while (runner) {
      if(runner.val===value) {return true;}
      runner = runner.next;
    }
    return false;
  }
  this.pushFront = function(value) {
    var oldHead = this.head;
    this.head = listNode(value);
    this.head.next = oldHead;
  }
  this.pushBack = function(value) {
    var newNode = new listNode(value);
    if (!this.head) { this.head = newNode;}
    else {
      var runner = this.head;
      while (runner.next)
      { runner = runner.next; }
      runner.next = newNode;
    }
  }
  this.popFront = function() {
    var returnVal = null;
    if (this.head) {
      returnVal = this.head.val;
      this.head = this.head.next;
    }
    return returnVal;
  }
  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.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;
  }
}
```

Week 8 - Lists II - Monday

CODING DOJO™

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

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

Each of the week's challenges has an *optional additional twist, if you so choose*. The challenges ask for a method in the *slList* class. To accept this optional challenge, solve each solution three ways: as standalone function, as *slNode* method, and as *slList* method (for *slNode* implementations, the node should treat itself as list head).

reverse

Create a method in the *slList* class that reverses the sequence of nodes in the list.

isPalindrome

Create a method to return whether the sequence of list values is a palindrome. With strings, palindromes read the same front-to-back and back-to-front. Here, compare node values as you would string characters. N.B.: to be accurate in JavaScript, we must use '===' instead of '==', since 1 == true == [1] == "1".

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

kthLast

Given k, return the value that is 'k' nodes from the list's end. Specifically if given 1, return the list's last value. If given 4, return the value at the node with exactly 3 nodes following it.

Tomorrow: dim sums and loopy loops....

Week 8 - Linked Lists II - Monday Recap

CODING DOJO™

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

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

reverse

```
function reverseList(headNode)
{
  var next, curr = headNode, prev = null;

  while (curr) {
    next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
  }
  return prev;
}
sList.prototype.reverse = function()
{ this.head = reverseList(this.head); }
slNode.prototype.reverse = function()
{ return reverseList(this); }
```

kthLast

```
function kthLastValue(headNode, k)
{
  if (k < 1) { return null; }

  var leader = headNode;
  while (leader && k >= 1) {
    k--;
    leader = leader.next;
  }
  if (k >= 1) { return null; }

  var follower = headNode;
  while (leader) {
    leader = leader.next;
    follower = follower.next;
  }
  return follower.val;
}
slNode.prototype.kthLastValue = function(k)
{ return kthLastValue(this, k);      }
sList.prototype.kthLastValue  = function(k)
{ return kthLastValue(this.head, k); }
```

isPalindrome

Second-level challenge: memory-frugal version.

```
function isPalindrome(head)
{
  var vals = [];
  while (head) {
    vals.push(head.val);
    head = head.next;
  }
  var idx = 0;
  while (idx < vals.length) {
    if (vals[idx] !== vals[vals.length-1]){
      return false;
    }
    vals.length--;
    idx++;
  }
  return true;
}
slNode.prototype.isPalindrome = function()
{ return isPalindrome(this);       }
sList.prototype.isPalindrome  = function()
{ return isPalindrome(this.head);  }

function isPalinMemFrugal(head)
{
  var runner = head, listLen = 0;
  while (runner) {
    listLen++;
    runner = runner.next;
  }
  var k = 1;
  while (k <= listLen / 2) {
    if (head.val !== kthLastValue(head,k)){
      return false;
    }
    head = head.next;
    k++;
  }
  return true;
}
```

Week 8 - Lists II - Tuesday

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

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

With minimal additional code, implement these as standalone functions as well as *slList* & *slNode* methods.

Sum List Numerals

Each node in two given lists has a value of 0-9 and signifies a digit in an overall number represented by that list. Sum the lists' numbers and return a new list representing this sum. First node represents least significant digit. Given (2)=>(0)=>(1) & (8)=>(4), return (0)=>(5)=(1) because 102 + 48 = 150.

Second-level: create another function where first node represents most significant digit.

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-level: also handle negative *shiftBy* (to left).

Setup List Loop

In preparation for tomorrow, create a sequence of slNodes that form a closed loop. The first parameter signifies how many nodes total, and the second parameter represents which node should be pointed to by the last node. Give nodes their node number as val, for clarity. Callling setupLoop(5, 3) should return a circular list of (1)=>(2)=>(3)=>(4)=>(5)=>(3)...

Tomorrow: Toucan Sam...

Week 8 - Linked Lists II - Tuesday Recap

CODING DOJO™

Sum List Numerals

Sum 2 lists' numbers; return a new list. First node represents least significant digit.
Second-level: first node represents *most* significant.

```
function sumNumsLSB(list1, list2) {
  var returnList, last = null;
  var sum = 0;
  while (list1 || list2 || sum) {
    if (list1) {
      sum += list1.val;
      list1 = list1.next;
    }
    if (list2) {
      sum += list2.val;
      list2 = list2.next;
    }
    var digit = slNode(sum % 10);
    if (last) { last.next = digit; }
    else { returnList = digit; }
    last = digit;
    sum = (sum / 10) % 1;
  }
  return returnList;
}
function sumNumsMSB(list1, list2) {
  var reverse1 = reverseList(list1);
  var reverse2 = reverseList(list2);
  var rSum = sumNumsLSB(reverse1,reverse2);
  return reverseList(rSum);
}

slNode.prototype.sumNumsMSB = function(list2)
{ return sumNumsMSB(this, list2); }
slNode.prototype.sumNumsLSB = function(list2)
{ return sumNumsLSB(this, list2); }
slNode.prototype.shiftList = function(shiftBy)
{ return shiftList(this, shiftBy); }

sList.prototype.sumNumsMSB = function(list2)
{ return sumNumsMSB(this.head, list2); }
sList.prototype.sumNumsLSB = function(list2)
{ return sumNumsLSB(this.head, list2); }
sList.prototype.shiftList = function(shiftBy)
{ this.head = shiftList(this.head, shiftBy); }
```

Shift List

Shift list nodes circularly to the right by *shiftBy*: when shifted off list's end, nodes reappear at list's start.
Second-level: handle negative *shiftBy* (to left).

```
function shiftList(head, shiftBy) {
  if (!head)     { return null; }
  var listLen = 0, runner = head, tail;
  while (runner) {
    listLen++;
    tail = runner;
    runner = runner.next;
  }
  shiftBy = Math.floor(shiftBy) % listLen;
  if (!shiftBy)    { return head; }
  if (shiftBy > 0) { shiftBy -= listLen; }
  var newTail = head;
  shiftBy++;
  while (newTail.next && shiftBy < 0) {
    newTail = newTail.next;
    shiftBy++;
  }
  var newHead = newTail.next;
  tail.next = head;
  newTail.next = null;
  return newHead;
}
```

Setup List Loop

```
function setupLoop(endNum, loopNum) {
  var firstNode = null, loopNode = null;
  var endNode, prevNode = null;
  if (loopNum===undefined) { loopNum=-1; }
  for (var nodeNum = 1; nodeNum <= endNum;
           nodeNum++) {
    var node = slNode(nodeNum);
    if (nodeNum==1)       { firstNode = node; }
    if (nodeNum==loopNum) { loopNode=node; }
    if (nodeNum==endNum)  { endNode = node;
      endNode.next = loopNode;
    }
    if (prevNode) { prevNode.next = node; }
    prevNode = node;
  }
  return firstNode;
}
```

Week 8 - Lists II - Wednesday

CODING DOJO™

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

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

Implement as standalone functions and *slList* & *slNode* methods. You may find setupListLoop() useful today.

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 that may or may not have a loop, return the total number of nodes. Given the circular list of (a)=>(b)=>(c)=>(d)=>(e)=>(c)=>(d)=>(e)... return 5.

Tomorrow: seeing double....

Week 8 - Linked Lists II - Wednesday Recap

CODING DOJO™

hasLoop

Determine whether linked list has a loop.

```
function hasLoop(head) {
  if (!head) { return false; }
  var runner = head.next, walker = head;
  while (runner && runner.next) {
    if (walker === runner) { return true; }
    walker = walker.next;
    runner = runner.next;
    runner = runner.next;
  }
  return false;
}
```

numNodes

List might have loop; return total number of nodes.

```
function numNodes(head)
{
  if (!head) { return 0; }
  var walker = head;
  var runner = head.next;
  var runCount = 1;

  while (runner && runner.next) {
    if (walker === runner) {
      walker = head;
      runner = runner.next;
      var walkCount = 1;

      while (walker.next != runner.next) {
        walker = walker.next;
        runner = runner.next;
        walkCount++;
      }
      while (walker != runner) {
        walker = walker.next;
        walkCount++;
      }
      return walkCount;
    }
    walker = walker.next;
    runner = runner.next;
    runner = runner.next;
    runCount += 2;
  }
  if (runner) { runCount++; }
  return runCount;
}
```

loopStart

Given list, return where loop begins, or null if none.

```
function loopStart(head) {
  if (!head) { return null; }
  var runner = head.next, walker = head;

  while (runner && runner.next) {
    if (walker === runner) {
      walker = head;
      runner = runner.next;
      while (walker != runner) {
        walker = walker.next;
        runner = runner.next;
      }
      return walker;
    }
    walker = walker.next;
    runner = runner.next;
    runner = runner.next;
  }
  return null;
}
```

breakLoop

If list has a loop, break it. Retain all nodes in order.

```
function breakLoop(head)
{
  if (!head) { return false; }
  var runner = head.next, walker = head;

  while (runner && runner.next) {
    if (walker === runner) {
      walker = head;
      runner = runner.next;
      while (walker.next != runner.next) {
        walker = walker.next;
        runner = runner.next;
      }
      runner.next = null;
      return true;
    }
    walker = walker.next;
    runner = runner.next;
    runner = runner.next;
  }
  return false;
}
```

Week 8 - Lists II - Thursday

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 data structure, all the same concepts and techniques as Singly Linked Lists still 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*

Given the reference implementations below for the doubly linked node and doubly linked list, can you construct the basic dList class? This would include dList methods push(),pop(), front(), back(), contains(), and size().
As a second challenge, implement these so that they are available as standalone functions as well as methods on the dlNode *and* dList classes.
As a gold-level challenge for the ambitious, implement functions we previously built for singly linked lists, such as prependValue(), kthToLastElement(), deleteMiddleNode(), isValid(), reverse(), isPalindrome(), partition().

```
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;
}
```

Week 8 - Linked Lists II - Thursday Recap

CODING DOJO™

DList Implementation - push/pop front/back contains size display

```
function dlPop(list) {
  if (list.tail) {
    var temp = list.tail.val;
    list.tail = list.tail.prev;
    if (list.tail) { list.tail.next=null; }
    else { list.head = null; }
    return temp;
  }
  return null;
}
dList.prototype.pop  = function()
{ return dlPop(this); }

function dlFront(list) {
  if (list.head) { return list.head.val; }
  return null;
}
dlNode.prototype.front = function() {
  var runner = this;
  while(runner.prev) {runner=runner.prev; }
  return runner.val;
}
dList.prototype.front  = function()
{ return dlFront(this); }

function dlContains(head, value) {
  var curr = head;
  while (curr) {
    if (curr.val == value) { return true; }
    curr = curr.next;
  }
  return false;
}
dlNode.prototype.contains = function(val)
{ return dlContains(this, val); }
dList.prototype.contains  = function(val)
{ return dlContains(this.head, val); }
dlNode.prototype.size = function()
{ return dlSize(this); }
dList.prototype.size  = function()
{ return dlSize(this.head); }
dlNode.prototype.display = function()
{ dlDisplay(this); return this; }
dList.prototype.display  = function()
{ dlDisplay(this.head); return this; }
```

```
function dlPush(list, val) {
  node = new dlNode(val);
  if (!list.head) { list.head = node; }
  else {
    list.tail.next = node;
    node.prev = list.tail;
  }
  list.tail = node;
  return this;
}
dList.prototype.push  = function(val)
{ dlPush(this, val); return this; }

function dlBack(list) {
  return list.tail ? list.tail.val : null;
}

dlNode.prototype.back = function() {
  var runner = this;
  while(runner.next) {runner=runner.next; }
  return runner.val;
}
dList.prototype.back  = function()
{ return dlBack(this); }

function dlSize(head) {
  var curr = head, count = 0;
  while (curr) {
    count++;
    curr = curr.next;
  }
  return count;
}

function dlDisplay(list) {
  var listStr = "[ ";
  if (list)
  { listStr += list.val; list = list.next;}
  while (list) {
    listStr += ", " + list.val;
    list = list.next; }
  listStr += " ]";
  console.log(listStr);
}
```

Week 8 - Lists II - Friday

CODING DOJO™

Take on as many challenges as possible, today and during the weekend!

Prepend Value

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

Append Value

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

Kth To Last Value

Given k, return the value 'k' from the list's end.

Delete Middle Node

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

Is Valid dList

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

Reverse

Create a function that reverses the nodes in a dList.

Palindrome

Determine whether a dList is a palindrome

Partition

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

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).

Break Loop

Given a dList that may contain a loop, break the loop while retaining original node order. Feel free to extend the function, making it a 'repair' partner for 'isValid'.

Week 9 - Sorts

CODING DOJO™

Overview

This week we investigate sorts, now that we have mastered both linked lists and recursion. Remember though that recursion is not an answer to every problem; everything solvable with recursion can also be solved without.

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 whether one piece of code is better than another? There are many different ways that one might assess software. For most people, their first thought is simply: *software is good when the features work*. That's a reasonable lithmus test, of course, but many aspects can go into *whether something works as expected.* In addition to basic functiionality, here are a few others: Is it resilient to unexpected inputs, or even malicious intent? Does it run in a variety of environments, such as different device form factors and/or different browsers? Is it trustworthy, regarding users' confidential data? Have message texts been factored appropriately so that the code can easily be localized into worldwide languages? Can it be easily configured and serviced by those that maintain and update it post-release? Is it accessible, for customers with visual or auditory disabilities? Is the source code generally understandable? (not just how deeply understood by one, but also how broadly understood across the team) How well documented is the source code? (or perhaps, the overall product) How rapidly / easily can it be extended to add new features? There are many dimensions of an excellent product.

In many situations, though, what trumps most of these factors is software *performance*. Again, this can mean many things, but the primary measures of software performance are run time and resource consumption (*time and space*, if you will). Everyone that works with software has experience with software that is frustratingly slow. Even small time optimizations can make a product feel more snappy and responsive to the user. Regarding resource consumption, this can be permanent storage (storage, database), memory (allocated "heap" space, call stack space), network bandwidth, or even power requirement -- after all, no one likes an app or web system that rapidly drains their mobile device's 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.

If we intend to compare the run times and memory consumptions of two piece of software, we need to be careful. What if one is written in PHP, and the other is written in C or even machine assembly language? What if one implementation is run on an Apple Watch, while the other is run on a massive IBM 16-processor server with 256 GB of RAM? To bend a metaphor, this would literally be comparing Apples and (Big) Blues. To factor out these differences and focus only on the algorithm itself, the analysis is measured in a relative sense, not an absolute one. Specifically, we compare the algorithm, in that particular language within that particular environment, to *itself*, when given different types and sizes of inputs. Specifically, by what percentage does our run time change when we double the size of the input? By what factor does our memory consumption grow, if we make our input ten times bigger? As input sizes get larger and larger, extraneous factors melt away, leaving only the 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 the "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 need to sort 4 billion numbers, then algorithm choice matters. Later this week, 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 week!

Week 9 - Sorts - Monday

CODING DOJO™

Bubble Sort array (review)

Create a function that, given an array of unsorted values, uses BubbleSort to sort the array in-place.

Selection Sort array (review)

Create a function that, given an array of unsorted values, uses SelectionSort to sort the array in-place.

Bubble Sort (list)

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

Tomorrow: Partition, merge

Week 9 - Sorts - Monday recap

CODING DOJO™

Bubble Sort array (review)

```
function bubbleSort(arr) {
  for(var end=0; end<arr.length-1; end++) {
    var run, lowV = arr[arr.length - 1];
    var swappedElements = false;
    for(run=arr.length-1; run>end; run--) {
      if (lowV < arr[run - 1]) {
        arr[run] = arr[run - 1];
        arr[run - 1] = lowV;
        swappedElements = true;
      }
      else { lowV = arr[run - 1]; }
    }
    if (!swappedElements) { break; }
  }
}
```

Selection Sort array (review)

```
function selectionSort(arr) {
  for(var idx=0; idx<arr.length-1; idx++) {
    var run, lowI=idx;
    for(run=idx+1; run<arr.length; run++) {
      if (arr[run]<arr[lowI]) { lowI=run; }
    }
    if (lowI != idx) {
      var temp = arr[lowI];
      arr[lowI] = arr[idx];
      arr[idx] = temp;
    }
  }
}
```

Bubble Sort (list)

```
function bubbleSortL(list) {
  var parent = node("parent");
  parent.next = list;

  var last = null;
  while (parent.next != last) {
    var runner = parent;
    var swappedElements = false;
    while (runner.next &&
           runner.next.next != last) {
      if (runner.next.val >
          runner.next.next.val) {
        var temp = runner.next;
        runner.next = temp.next;
        temp.next = runner.next.next;
        runner.next.next = temp;
        swappedElements = true;
      }
      runner = runner.next;
    }
    last = runner.next;
    if (!swappedElements) { break; }
  }
  return parent.next;
}
```

Selection Sort (list)

```
function selectionSortL(list) {
  if (!list || !list.next) { return list; }
  var parent = node("parent");
  parent.next = list;

  var slow = parent;
  while (slow.next.next) {
    var lowest = slow, fast = slow;
    var lowVal = slow.next.val;
    while (fast.next) {
      if (fast.next.val < lowVal) {
        lowVal = fast.next.val;
        lowest = fast;
      }
      fast = fast.next;
    }
    if (lowest != slow) {
      var lowNode = lowest.next;
      lowest.next = lowest.next.next;
      lowNode.next = slow.next;
      slow.next = lowNode;
    }
    slow = slow.next;
  }
  return parent.next;
}
```

Week 9 - Sorts - Tuesday

CODING DOJO™

Concept: Big-O Notation

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

(Quick side note: the 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 indicates *best-case;* Big-Theta *average case*. Their values are a little different, but in many purposes you could think of them as the same. Further, when best-case and worst-case differ, most people just talk about Big-O and perhaps also "best-case".)

What does Big-O notation indicate? It conveys how the algorithm will perform, as the input sizes grow very large. In other words, as we multiply the size of the input by some factor N, by what factor does the run time change; by what factor does the memory consumption change? In *practice*, we would measure it this way: first give an input of a specific size, and measure how long it took to run and/or how much memory was required; then give a similar input of "size x N", and compare the run time and memory needed -- this ratio, in terms of N, is our Big-O.

For example, consider a `FindMax()` algorithm to return lowest value in an unsorted array. If we double the array size, we expect the algorithm to run twice as long. In other words, if we multiply the array length by N, run time should be multiplied 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 memory complexity of this algorithm 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). Get it? One last thing: with recursion, we also factor in memory consumed by additional stack space as we recurse. We'll briefly touch on that later.

Insertion Sort (review)

Create a function that, given an array of unsorted values, uses InsertionSort to sort the array in-place. What is the run-time complexity of InsertionSortArr? What is the space complexity of InsertionSortArr?

Partition Array

Create a function that, given an unsorted array, partitions in-place. Use first element as pivot; return index where pivot ended up. For [5, 1, 8, 4, 9, 2, 5, 3], change array to [1, 4, 2, 3, 5, 8, 9, 5] and return 4.

Second-level challenge: Be clever selecting pivot; use the median of the *first*, *last* and *middle*.

Third-level challenge: Partition an array *subset*, with parameters *start* and *end*. Exclude end; hence default values for start and end are 0 and array.length.

Insertion Sort (list)

Create a function that uses InsertionSort to sort singly linked lists. The list nodes contain .val and .next, as well as other attributes that you should not reference. What are the run-time and space complexities?

Partition List (review)

Create a function that partitions a singly linked list. Use the first element as pivot, and return the new list. List nodes contain .val and .next, plus other attributes you should not reference. For example, given singly linked list { 5=>1=>8=>4=>9=>2=>5=>3 }, return the list { 1=>4=>2=>3=>5=>8=>9=>5 }.

Tomorrow: Sort this, quick, before the lava reaches the village....

Week 9 - Sorts - Tuesday recap

CODING DOJO™

```
// O(n2) run-time, O(1) space
function insertionSort(arr) {
  for (var idx=1; idx<arr.length; idx++) {
    var lowVal = arr[idx];
    var changed = false;
    for (var run = idx-1; run>=0; run--) {
      if (arr[run] > lowVal) {
        arr[run + 1] = arr[run];
        changed = true;
      }
      else { break; }
    }
    if (changed) { arr[run + 1] = lowVal; }
  }
}
```

```
function partitionL(head, value) {
  var leftEnd, leftStart = null;
  var rightEnd, rightStart = null;
  var match = null, runner = head;
  while (runner) {
    if (runner.val === value && !match)
    { match = runner; }
    else if (runner.val < value) {
      if (!leftStart)  { leftStart = runner;  }
      else             { leftEnd.next=runner; }
      leftEnd = runner;
    } else {
      if (!rightStart) { rightStart = runner;  }
      else             { rightEnd.next=runner; }
      rightEnd = runner;
    }
    runner = runner.next;
  }
  if (rightEnd) { rightEnd.next = null; }
  head = rightStart;
  if (match) {
    match.next = head;
    head = match;
  }
  if (leftStart) {
    leftEnd.next = head;
    head = leftStart;
  }
  return head;
}
```

```
// O(n2) run-time, O(1) space
function insertionSortL(list) {
  if (!list || !list.next) { return list; }
  var parent = node("parent");

  while (list) {
    var firstNode = list;
    list = list.next;
    sortedInsert(parent, firstNode);
  }
  return parent.next;
}
function sortedInsert(list, node) {
  while (list.next && list.next.val<=node.val)
  { list = list.next; }
  node.next = list.next;
  list.next = node;
}
```

```
function partitionA(arr, start, end) {
  start = start || 0;
  end = end || arr.length;
  var pivotIdx = pickPivot(arr, start, end);
  var pivot = arr[pivotIdx];
  arr[pivotIdx] = arr[start];
  var lastLowIdx = start;

  for (var idx = start + 1; idx < end; idx++) {
    if (arr[idx] < pivot) {
      var temp = arr[idx];
      arr[idx] = arr[++lastLowIdx];
      arr[lastLowIdx] = temp;
    }
  }
  arr[start] = arr[lastLowIdx]
  arr[lastLowIdx] = pivot;
  return lastLowIdx;
}
function pickPivot(arr, start, end) {
  var a = arr[start], c = arr[end - 1];
  var bI =Math.floor((start+end)/2), b=arr[bI];
  return (((a-b)*(b-c) >= 0) ? bI
       : (((a-c)*(c-b) >= 0) ? end-1 : start));
}
```

Week 9 - Sorts - Wednesday

CODING DOJO™

In talking about sorting algorithms, we have talked mostly about how we measure the *run-time* performance of an algorithm – *how does the time needed to run this algorithm change, as the size of the input grows large?* However, Big-O doesn't refer only to runtime; it can also refer to any resource consumed, such as memory, storage or even network bandwidth. Other than runtime, the most common resource tracked in algorithms is RAM. If an algorithm consumes a lot of RAM, it uses either a lot of heap, or a lot of call-stack, or both. These correspond to 1) making a copy of the input or 2) making a significant number of recursive function calls. Clearly it is much better, all other things equal, if an algorithm doesn't have to make a copy of the input data. After all, what if we are sent an array containing 4 billion values? Also, call-stacks are not unlimited in size, so it doesn't take much to "blow our stack" so to speak. Remember that everything solvable with recursion can also be solved without. More on this, and Big-O of space in general, in a few days.

So, really, which sorting algorithm is best? Again, there is no single correct answer, other than "depends on the 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 week. Today we discuss what makes an algorithm *Adaptive*.

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

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

When is this important? Consider if you have a huge quantity of existing, already-sorted data, and you need to add in a small number of new, unsorted values. With InsertionSort, we could place the new values after the existing data, and quickly sort them into place with little penalty from the magnitude of existing data. Big win!

QuickSort (array)

Create a function that, given an array of unsorted values, uses yesterday's PartitionArray to sort the 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

Create a function that combines two already-sorted arrays, returning a new sorted array with all elements.

Partition3

With our previous PartitionArr implementation, any duplicates of the pivot were not grouped together. This makes the algorithm cleaner, but performs poorly in the specific case of arrays containing many duplicate values. Create partition3() that partitions an unsorted array; keep duplicate pivot elements together; *return a two-element array containing the first pivot value and the first greater value*. For [5, 1, 8, 4, 9, 2, 5, 3], change array to [1, 4, 2, 3, 5, 5, 8, 9] and return [4, 6]. Note: the other 5 moved forward, next to pivot.

Second-level challenge: Pick a more optimal pivot.

Third-level challenge: Add the ability to partition only a *portion* of the given array, with *start* and *end*.

Tomorrow: A perfect merger of truth and beauty

Week 9 - Sorts - Wednesday recap

CODING DOJO™

QuickSort Array

```
// default: use 0 and full arr length
function quickSort(arr, start, end) {
  start = start || 0;
  end = end || arr.length;

  if (end - start <= 1)  { return; }
  var mid = partition(arr, start, end);
  quickSort(arr, start, mid);
  quickSort(arr, mid + 1, end);
}
```

Combine Sorted Arrays

Create a function that combines two already-sorted arrays, returning a new sorted array with all elements.

```
function combine(arr1, arr2) {
  var arr3 = [];
  var idx1 = 0, idx2 = 0;
  while (idx1 < arr1.length
      && idx2 < arr2.length) {
    if (arr1[idx1] <= arr2[idx2]) {
      arr3.push(arr1[idx1++]);
    } else {
      arr3.push(arr2[idx2++]);
    }
  }
  while (idx1 < arr1.length) {
    arr3.push(arr1[idx1++]);
  }
  while (idx2 < arr2.length) {
    arr3.push(arr2[idx2++]);
  }
  return arr3;
}
```

Partition3

With our previous PartitionArr implementation, any duplicates of the pivot were not grouped together. This makes for a cleaner algorithm overall, but poor performance in a specific case of arrays containing many duplicate values. Create partition3() that partitions an array of unsorted values; keep duplicate pivot elements together; return a two-element array containing the indices of the first pivot and the first greater value. Ex.: change input [5, 1, 8, 4, 9, 2, 5, 3] to [1, 4, 2, 3, 5, 5, 8, 9] and return [4, 6]. Note: the other 5 moved forward, next to pivot.

Second-level challenge: Pick a more optimal pivot.

Third-level: Partition an array *portion*, given *start/end*.

```
function partition3(arr, start, end) {
  start = start || 0;
  end = end || arr.length;
  var pivotIdx = pickPivot(arr,start,end);
  var pivot = arr[pivotIdx];
  arr[pivotIdx] = arr[start];
  var lastLowIdx=start, firstHighIdx=end;

  for (var idx = start + 1;
       idx < firstHighIdx; idx++) {
    if(arr[idx]<pivot&&(lastLowIdx+1<idx)){
      var temp = arr[idx];
      arr[idx] = arr[++lastLowIdx];
      arr[lastLowIdx] = temp;
    } else {
      if (arr[idx] > pivot) {
        var temp = arr[--firstHighIdx];
        arr[firstHighIdx] = arr[idx];
        arr[idx--] = temp;
      }
    }
  }
  arr[start] = arr[lastLowIdx]
  arr[lastLowIdx] = pivot;
  return [lastLowIdx, firstHighIdx];
}
```

Week 9 - Sorts - Thursday

CODING DOJO™

We've discussed what makes a sorting algorithm *Adaptive*. Can it sometimes run much 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?

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).

Overview

This week we investigate sorts, now that we have mastered both linked lists and recursion. Remember though that recursion is not an answer to every problem; everything solvable with recursion can also be solved without.

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.

MergeSort (array)

Use yesterday's CombineArrs function above to construct mergeSortA() for an unsorted array.
What is the run-time complexity of MergeSortArr?
What is the space complexity of MergeSortArr?
Is MergeSortArr *stable*?

MergeSort (list)

Use CombineLists from yesterday to construct the MergeSortL algorithm for an unsorted singly linked list.
What is the run-time complexity of MergeSortList?
What is the space complexity of MergeSortList?
Is MergeSortList *stable*?

Tomorrow: Tricks and treats

Week 9 - Sorts - Thursday recap

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.

MergeSort (list)

Use the CombineLists function above to construct the MergeSort algorithm for an unsorted singly linked list.
What is the run-time complexity of MergeSortList?
What is the space complexity of MergeSortList?

MergeSort (array)

Use the CombineArrs function above to construct the MergeSort algorithm for an unsorted array.
What is the run-time complexity of MergeSortArr?
What is the space complexity of MergeSortArr?

```
function mergeSort(arr)
{
  if (arr.length < 2) { return arr; }

  var arr2 = arr.splice(0, arr.length / 2);
  arr = mergeSort(arr);
  arr2 = mergeSort(arr2);
  return combine(arr, arr2);
}
```

Week 9 - Sorts - Friday

-

Overview

This week we investigate sorts, now that we have mastered both linked lists and recursion. Remember though that recursion is not an answer to every problem; everything solvable with recursion can also be solved without.

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 complexity of CountingSort?
What is the space complexity of CountingSort?

QuickSort3

Create a QuickSort3 function that uses Partition3 to sort an array in-place.

This weekend: Get radical with radix sort...

Week 10 - Binary Search Tree 1

CODING DOJO™

Overview

This week we begin an exploration of the **binary search tree**, a very important data structure. As we will see, the binary search tree (BST) is optimized for quickly finding and retrieving elements. A BST is similar to a linked list, in that it stores data elements within node objects. Let's further compare a *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; with a linked list we call it the head, with a binary tree we call it the root. Finally, a binary tree is similar to a linked list in that no specific order must be imposed, as far as where values must be located within it.

Any binary tree node can have a left child and/or a right child, and each of those children might have left and/or right children of their own. Just as an entire section of a family tree might descend from one brother as opposed to another, similarly there are related subsets of a binary tree. These are (not surprisingly) called subtrees. We refer to all nodes stemming from the root node's left pointer as the root's left subtree, for example.

The **binary search tree** simply adds a requirement that for a particular node, all the nodes in its left subtree must have values that are less than its own. Similarly on the right side, every value in the subtree underneath a node's right pointer must be greater than or equal to that node's value. Remember that this constraint holds for every node in the subtree, not just the direct child. Accordingly, this determines exactly where new children are placed in a BST. If "Grandparent" node 50 has a right child with value 75, then new children of node 75 are appropriately constrained. Specifically, to add a node with value 60 we would make it node 75's left child, but we could not do this if the node to be added had value 45. (We would place node 45 in "grandparent" node 50's *left* subtree.)

BST nodes furthest from the root are considered *leaf* nodes: those without children. Depending on the 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 (if that root node contains the smallest or largest value in the tree, respectively).

A tree's **depth** is the length from root to farthest leaf, including both nodes. If you add nodes to a binary search tree in randomized order, the tree grows in a relatively *balanced* manner – left and right subtrees will be about the same size, and mostly equal depth across the BST. If you add elements in *sorted* order, the tree will become *unbalanced*, resembling a linked list in shape, and the depth might approach the total number of elements.

Even balanced trees might still have "holes" with no nodes. The fewer holes, the more **full** it is - a measurement of the tree's "bushiness". Given two trees with the same number of nodes, the shorter will always be more full (or bushy), and that's a good thing when it comes to BSTs. More nodes, in the same depth, is more efficient.

Finding values in BSTs should be vastly faster than finding them in SLists. BSTs are ordered, so you quickly narrow in on a range of closely related values. Instead of searching all values, the longest search is only as far as the depth of the tree. This is why full, shallow trees are best. As data structures go, BSTs *rock* for fast retrieval.

Week 10 - Binary Search Tree 1 - Monday

This week's BST challenges will be based on the following reference definitions:

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

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

add(value)

Create a 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(value)

Create a 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.

isEmpty()

Create a method returning whether the bst is empty (an empty bst contains no values).

Week 10 - Binary Search Tree 1 - Monday recap

CODING DOJO™

This week's BST challenges are based on the following reference definitions:

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

```
function bst() {
  this.root = null;
}
// Tomorrow we'll add a pattern to both
// objects to make them callable w/out new
```

add(value)

Create a 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.

```
this.add = function(value)
{
  var node = btNode(value);
  if (!this.root) {
    this.root = node;
  }

  var runner = this.root;
  while (runner !== node) {
    if (value < runner.val) {
      if (!runner.left) {
        runner.left = node;
      }
      runner = runner.left;
    }
    else {
      if (!runner.right) {
        runner.right = node;
      }
      runner = runner.right;
    }
  }
}
```

contains(value)

Create a 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.

```
this.contains = function(value)
{
  var runner = this.root;
  while (runner) {
    if (runner.val === value)
    { return true; }

    if (value < runner.val)
    { runner = runner.left; }
    else
    { runner = runner.right; }
  }
  return false;
}
```

isEmpty()

Create a method returning whether the bst is empty (an empty bst contains no values).

```
// Easy peezy!
this.isEmpty = function()
{ return (this.root === null); }
```

Tip: interviewers refering to *Tree* or *Binary Tree* almost always mean *Binary Search Tree*. Always good to clarify.

Week 10 - Binary Search Tree 1 - Tuesday

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

min()

Create a method on the bst class that returns the smallest value found in the bst.

max()

Create a bst method to return the bst's largest value.

size()

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

isValid()

Construct a bst method to determine whether a tree has valid structure. Specifically, ensure nodes/values are appropriately located in left or right subtrees.

Week 10 - Binary Search Tree 1 - Tuesday recap

CODING DOJO™

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

min()

Return the smallest value in the BST.

```
bst.prototype.min = function()
{
  if (this.isEmpty()) { return null; }
  var runner = this.root;
  while (runner.left) {runner=runner.left;}
  return runner.val;
}
```

max()

Create a bst method to return the BST's largest value.

```
bst.prototype.max = function()
{
  if (this.isEmpty())  { return null; }
  var runner = this.root;
  while(runner.right){runner=runner.right;}
  return runner.val;
}
```

size()

Return the number of nodes contained in the tree.

```
bst.prototype.size = function() {
  if (!this.root) { return 0; }
  return this.root.size();
}
```

```
btNode.prototype.size = function()
{
  var size = 1;
  if (this.left)  {
    size += this.left.size();
  }
  if (this.right) {
    size += this.right.size();
  }
  return size;
}
```

isValid()

Ensure values are located in appropriate subtrees.

```
bst.prototype.isValid = function() {
  if (!this.root)
  { return true;  }
  if (!(this.root instanceof btNode)
  { return false; }
  return this.root.isValid();
}
```

```
btNode.prototype.isValid =
function(minBound, maxBound)
{
  if (minBound===undefined)
  { minBound = -Number.MAX_VALUE; }
  if (maxBound===undefined)
  { maxBound =  Number.MAX_VALUE; }

  if (this.val< minBound) { return false; }
  if (this.val>=maxBound) { return false; }

  var valid = true;
  if (this.left)
  { valid = valid &&
    this.left.isValid(minBound, this.val);}
  if (this.right) {
    valid = valid &&
    this.right.isValid(this.val,maxBound);}
  return valid;
}
```

Week 10 - Binary Search Tree 1 - Wednesday

CODING DOJO™

Today implement the challenges in the BST class below. Notice anything new?

```
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;
}
```

remove(value)

Remove given value. Return true if found, false if not.

removeAll()

Clear all values from the tree.

addWithoutDupes(value)

Add given value *only if it is not already found*.
Remembering Set theory, this changes our BST from multiset to set. Return true if added, false otherwise.

Week 10 - Binary Search Tree 1 - Wednesday recap

CODING DOJO™

```
function bst() {
  if (!(this instanceof bst))
  { return new bst(); }
  this.root = null;
}
function btNode(value) {
  if (!(this instanceof btNode))
  { return new btNode(value); }
  this.val = value;
  this.left = null;
  this.right = null;
}


bst.prototype.removeAll = function()
{ this.root = null; }
// C'est tout!  Magnifique!


bst.prototype.addNoDupes = function(value){
  var runner = this.root;
  while (runner) {
    if (value===runner.val) {return false;}
    if (value < runner.val) {
      if (!runner.left)
      {
        runner.left = btNode(value);
        return true;
      }
      runner = runner.left;
    } else {
      if (!runner.right)
      {
        runner.right = btNode(value);
        return true;
      }
      runner = runner.right;
    }
  }
  this.root = btNode(value);
  return true;
}
```

```
bst.prototype.remove = function(value) {
  if (!this.root) { return false; }
  var nodeToDel = null, parent = this.root;
  if (this.root.val === value) // remove root?
  { nodeToDel = this.root; }
  while (!nodeToDel) {    // find node to remove
    if (parent.val <= value) {
      if (!parent.right) { return false; }
      if (parent.right.val === value)
           { nodeToDel = parent.right; }
      else { parent    = parent.right; }
    } else {
      if (!parent.left)  { return false; }
      if (parent.left.val === value)
           { nodeToDel = parent.left; }
      else { parent    = parent.left; }
    }
  }
  if (!nodeToDel.left) {      // "easy" removes?
    if (nodeToDel == this.root)
         { this.root    = nodeToDel.right; }
    else if (parent.left == nodeToDel)
         { parent.left  = nodeToDel.right; }
    else { parent.right = nodeToDel.right; }
  } else if (!nodeToDel.right) {
    if (nodeToDel == this.root)
         { this.root    = nodeToDel.left; }
    else if (parent.left == nodeToDel)
         { parent.left  = nodeToDel.left; }
    else { parent.right = nodeToDel.left; }
  } else {     // "hard" remove (has 2 children)
    var follower = nodeToDel;
    var runner = nodeToDel.right;
    while (runner.left) {
      follower = runner;
      runner = runner.left;
    }       // runner is nodeToDel's replacement
    runner.left = nodeToDel.left;
    if (nodeToDel == this.root) //connect parent
         { this.root    = runner; }
    else if (nodeToDel == parent.left)
         { parent.left  = runner; }
    else { parent.right = runner; }
    if (follower != nodeToDel) { //other cleanup
      follower.left = runner.right;
      runner.right = nodeToDel.right;
    }
    return true;
}
```

Tomorrow: Climbing back and forth in our trees

Week 10 - Binary Search Tree 1 - Thursday

CODING DOJO™

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

height()

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

isBalanced()

Write a 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.

›morrow: Climbing back and forth in our trees

Week 10 - Binary Search Tree 1 - Thursday recap

CODING DOJO™

height()

Build a method on the bst object that returns the total height of the tree – the longest sequence of nodes from root node to leaf node. To accomplish this, it may be helpful to create a btNode.height() method.

```
bst.prototype.height = function()
{
  return ( this.root)
         ? this.root.height()
         : 0;
}

btNode.prototype.height = function()
{
  var lHeight = ( this.left)
                ? this.left.height()
                : 0;
  var rHeight = ( this.right)
                ? this.right.height()
                : 0;

  return Math.max(lHeight, rHeight) + 1;
}
```

isBalanced()

Write a 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.

```
bst.prototype.isBalanced = function()
{
  return (  this.root)
         ? (this.root.balHeight() >= 0)
         : true;
}

btNode.prototype.balHeight = function()
{
  var lHeight = ( this.left)
                ? this.left.height()
                : 0;
  var rHeight = ( this.right)
                ? this.right.height()
                : 0;

  if (  lHeight == -1
     || rHeight == -1
     || Math.abs(lHeight - rHeight) > 1)
  {
    return -1;
  }

  return Math.max(lHeight, rHeight) + 1;
}
```

This weekend: talk to a few trees (do they listen to you?)

Week 10 - Binary Search Tree 1 - Friday

Add these methods that use in-order traversal. You may 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.

valBefore(value)

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

valAfter(value)

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.

Week 10 - Binary Search Tree 1 - Friday recap

CODING DOJO™

Add these bst methods, using in-order traversal. You may need an attribute .parent.

valBefore(value)

Given a value that may not be in the tree, return the next smaller value. For {2, 5, 8} valBefore(3) returns 2. If you add .parent, what other methods must change?

```
bst2.prototype.valBefore = function(value){
  var found = null;
  if (!this.root) { return null; }

  var runner = this.root;
  while (runner.val !== value) {
    if (runner.val > value) {
      if (!runner.left) { break; }
      runner = runner.left;
    }
    else {
      if (!runner.right) { break; }
      runner = runner.right;
    }
  }
  if (runner.left) {
    runner = runner.left;
    while (runner.right)
    { runner = runner.right; }
  } else {
    while (runner && runner.val >= value)
    { runner = runner.parent; }
    if (!runner) { return null; }
  }
  return runner.val;
}
```

valAfter(value)

Given a value that may not be in the tree, return the next larger value. For {2, 5, 8} valAfter(8) returns null. If you add .parent, what other methods must change?

```
bst2.prototype.valAfter = function(value) {
  var found = null;
  if (!this.root) { return null; }

  var runner = this.root;
  while (runner.val !== value) {
    if (runner.val < value) {
      if (!runner.right) { break; }
      runner = runner.right;
    }
    else {
      if (!runner.left) { break; }
      runner = runner.left;
    }
  }
  if (runner.right) {
    runner = runner.right;
    while (runner.left)
    { runner = runner.left; }
  } else {
    while (runner && runner.val <= value)
    { runner = runner.parent; }
    if (!runner) { return null; }
  }
  return runner.val;
}
```

```
// bst methods add(), isValid(), remove() and addNoDupes(), btNode method isValid(), and
// both constructors must change to incorporate .parent into btNode. For example,
// * btNode constructor would include "this.parent = null;"
// * btNode.isValid includes "if (this.left && this.left.parent !== this){ return false; }"
// * bst.isValid adds "if (this.root.parent) { return false; }"
// * in bst.add after creating a node and adding at runner, "newNode.parent = runner;"
// * bst.addNoDupes: same.
// * in bst.remove if we move another node in place of the removed node, we set up parent
//   ptrs for the moved node as well as that node's new children.

// Work out all the changes yourself!  Also, see the next page for more weekend fun.
```

Week 10 - Binary Search Tree 1 - Weekend

This weekend, create these additional standalone functions to manage BSTs:

bst2Arr()

Create a standalone function that accepts a bst object and outputs a new array containing the values of the BST, traversed in-order.

Second-level challenge: create bst2ArrPre and bst2ArrPost() that return arrays of the BST's values, traversed pre-order and post-order respectively.

Third-level challenge: refactor to minimize code.

bst2List()

Create a standalone function that accepts a bst object and outputs a singly linked list containing the values of the BST, traversed in-order.

Second-level challenge: create bst2ListPre and bst2ListPost() that return sLists of the BST's values, traversed pre-order and post-order respectively.

Third-level challenge: refactor to minimize code.

Next week: westward ho, to new data structures...

Week 11 - Data Structures 1

CODING DOJO™

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 week 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.

Week 11 - Data Structures 1 - Monday

CODING DOJO™

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 our 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	**contains(val)**:	does queue contain given val?
dequeue():	remove & return front value	**isEmpty()**:	does queue contain anything?
front():	return (not remove) front value	**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) that adds the given value to end of our queue. Remember, slQueue uses a singly linked list (not an array).

Dequeue

Create slQueue method dequeue() that removes and returns value at front of queue. Remember, slQueue class uses a singly linked list (not array).

Front

Create slQueue method front() that returns the value at front of our queue, without removing it.

Contains

Create slQueue method contains(val) that returns whether the given value is found within our queue.

Is Empty

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

Size

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

Tomorrow: Growing / shrinking the map

Week 11 - Stacks and Queues - Monday recap

CODING DOJO™

Here are slQueue and node implementations, after yesterday's challenges:

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

  this.enqueue = function(value){
    var newNode = node(value);
    if (head) { tail.next=newNode;}
    else      { head     =newNode;}
    tail = newNode;
  }

  this.front = function() {
    if (head) { return head.val; }
    return null;

/*  or the one-liner ...
    return head ? head.val : null;
*/
  }

  this.dequeue = function() {
    if (!head) { return null; }
    var returnVal = head.val;
    head = head.next;
    if (!head) { tail = null; }
    return returnVal;
  }

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

  this.size = function(){
    var count = 0;
    var runner = head;
    while (runner) {
      count++;
      runner = runner.next;
    }
    return count;
  }

  this.isEmpty = function() {
    return (head === null);
  }
}


// node implementation is unchanged
function node(value) {
  if (!(this instanceof node))
  {   return new node(value); }

  this.val  = value;
  this.next = null;
}
```

Week 11 - Stacks and Queues - Tuesday

CODING DOJO™

Circular Queues

Because of the one-way direction of singly linked lists, using one to implement a Queue feels very natural. What if we wanted to implement a Queue using an underlying *Array*? Why would we do this? In certain high-performance scenarios, we may not be able to allocate memory while working with our queue. We may need a Queue object that does not create a new node object to add a value. Although we could always 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 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!

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!

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

Enqueue

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

Dequeue

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

Front

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

Contains

Return whether given val is within the queue.

Is Empty

Return whether queue is empty.

Size

Return number of queued vals (not capacity).

Is Full

Return whether queue is full.

Grow

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

Tomorrow: I'll take a short stack....

Week 11 - Stacks and Queues - Tuesday recap

Here is the cirQueue implementation from yesterday's challenges:

```
function cirQueue(size) {
  if (!(this instanceof cirQueue))
  { return new cirQueue(qCapacity);}
  var headIdx = 0;
  var tailIdx = 0;
  var buffer = [];
  var capacity = qCapacity;
  var numVals = 0;

  this.enqueue = function(value) {
    if (this.isFull())
    { return false; }

    buffer[tailIdx++] = value;
    tailIdx %= capacity;
    numVals++;
    return true;
  }
  this.dequeue = function() {
    if (this.isEmpty())
    { return null; }

    var val = buffer[headIdx++];
    headIdx %= capacity;
    numVals--;
    return val;
  }
  this.front = function() {
    if this.isEmpty())
    { return null; }
    return buffer[headIdx];
  }
  this.isEmpty = function() {
    return (numVals === 0);
  }
  this.isFull = function() {
    return (numVals === capacity);
  }
  this.contains = function(value) {
    for (var count=0; count<numVals;
         count++) {
      var idx = (headIdx + count)
      idx %= capacity;
      if (buffer[idx] === value)
      { return true; }
    }
    return false;
  }
  this.size = function() {
    return numVals;
    // capacity+tail-head)%capacity;
  }
  this.grow = function(newCapacity){
    if (newCapacity < capacity)
    { return false; }
    if (headIdx >= tailIdx) {
      var newIdx;
      for (var idx = 0;
           idx < tailIdx; idx++) {
        newIdx = (capacity + idx);
        newIdx %= newCapacity;
        buffer[newIdx]=buffer[idx];
      }
    }
    tailIdx += capacity;
    tailIdx %= newCapacity;
    capacity = newCapacity;
  }
}
```

Week 11 - Stacks and Queues - Wednesday

CODING DOJO™

Stacks and Deques

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.

Stacks can be easily and rapidly implemented with Arrays. Let's build one with an sList instead.
Implement the following methods on an sList-based Stack object:

Push

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

Pop

Create pop() that removes & returns the top val.

Top

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

Contains

Return whether given val is within the stack.

Is Empty

Return whether the stack is empty.

Size

Return the number of stacked values.

Stack / Queue Inheritance

As an exercise in object-oriented design, combine the slQueue you wrote on Monday with the slStack you just created. To help in consolidation, in your the Queue object use Stack method names push/pop/top instead of enqueue/dequeue/front). The combined code of two classes should be only about 30% larger than the code of one! Which class inherits from which? Or is there a parent class that is neither?

Deque Class Implementation

Having combined the Stack and Queue classes *codewise*, 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, based on any previous stack or queue implementation, containing methods pushFront(val), pushback(val), popFront(), popBack(), front(), back(), contains(val), isEmpty(), and size().

Tomorrow: the crown prince of data structures!

Week 11 - Stacks and Queues - Wednesday recap

CODING DOJO™

Here are the **slStack**, **slQueue** and **deque** implementations from yesterday:

```
function slNode(value) {
  this.next = null;
  this.val = value;
}
function slStack() {
  this.slTop = null;
}

slStack.prototype.push = function(value) {
  var node = slNode(value);
  node.next = this.slTop;
  this.slTop = node;
}

slStack.prototype.pop = function() {
  if (!this.slTop) { return null; }
  var returnVal = this.slTop.val;
  this.slTop = this.slTop.next;
  return returnVal;
}

slStack.prototype.top = function() {
  if (!this.slTop) { return null; }
  return this.slTop.val;
}

slStack.prototype.isEmpty =
function(){ return (!this.slTop); }

slStack.prototype.size = function() {
  var count = 0;
  var runner = this.slTop;
  while (runner) {
    count++;
    runner = runner.next;
  }
  return count;
}

slStack.prototype.contains = function(value) {
  var runner = this.slTop;
  while (runner) {
    if (runner.val===value) { return true;}
    runner = runner.next;
  }
  return false;
}
```

```
function slQueue() {
  slStack.call(this);
  this.slBack = null;
}
slQueue.prototype =
              Object.create(slStack.prototype);
slQueue.prototype.constructor = slQueue;
slQueue.prototype.push = function(value) {
  var newNode = slNode(value);
  if (this.slTop)
  { this.slBack.next = newNode; }
  else  { this.slTop = newNode; }
  this.slBack = newNode;
}

function deque(capacity) {
  cirQueue.call(this, capacity);
}
deque.prototype =
             Object.create(cirQueue.prototype);
deque.prototype.constructor = deque;
deque.prototype.pushBack = function(value)
{ return this.enqueue(value); }
deque.prototype.pushFront =f unction(value) {
  if (this.isFull()) { return false; }
  this.headIdx += (this.capacity - 1);
  this.headIdx %= this.capacity;
  this.buffer[this.headIdx] = value;
  this.numVals++;
  return true;
}
deque.prototype.popFront = function()
{ return this.dequeue(this); }
deque.prototype.popBack = function() {
  if (this.isEmpty()) { return null; }
  this.tailIdx += (this.capacity - 1);
  this.tailIdx %= this.capacity;
  var val = this.buffer[this.tailIdx];
  this.numVals--;
  return val;
}
deque.prototype.back = function() {
  if (this.isEmpty()) { return null; }
  return this.buffer[(this.tailIdx - 1 +
this.capacity) % this.capacity]; }
```

Week 11 - Stacks and Queues - Thursday

CODING DOJO™

Priority Queues

Queues and Stacks are optimized for quick addition – and quick removal, so long as you want values extracted in arrival order (or *reverse* order, for Stacks). They are *not* optimized for search: elements are stored linearly, without regard for the values. We might iterate every contained value before finding what we seek (for example).

What if instead we created a data structure that acted like a Queue, but *did* care about values instead of insertion time. With this data structure, we could insert elements in any sequence, but it would maintain the elements in value order, so that the lowest value was always first in line to be read. After inserting values in any fashion, we could then extract (like pop or dequeue) one element at a time, and always get the lowest value. This would 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 (assuming we rank items based on priority). In fact, OS subsystems such as networking and storage work in exactly 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 examine how to build a *Priority Queue*.

Sequencer

Build sequenceMessage and playMessages. 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.

According to legend, a mythical beast called the *manticore* had a lion's body, a human head, and a scorpion's tail. The most common *priority queue* implementation is called a *minheap* and has similarly unusual characteristics. (Note: this *heap* data structure is different than the heap where memory allocations occur, although regrettably they use the same term.) Not to be outdone by the manticore, a heap behaves like a queue, internally manages data like a binary tree, and is stored in an array. As a result, instead of extraordinary speeds in one category at the expense of poor speeds elsewhere, heaps strike a balance: great insertion, good deletion, and good extraction (so long as we want items extracted in priority order). So, how does this creature do it?

Interestingly, a heap isn't fully sorted. It is sorted "just enough" to immediately extract the lowest value then quickly rearrange other values, so that it is again sorted "just enough" for the next extraction. Similarly, insertion keeps the tree "just sorted enough", without doing extra work. This laziness actually leads to high performance. (Here we assume a *minheap*, but the rules are easily inverted for maxheaps: highest values first, not lowest.) First, data in a heap are arranged in binary nodes. Second, every heap node must have a value less than or equal to its children. Third, the nodes form a *complete* binary tree, where every node has two children except at the deepest level, where nodes are present starting from leftmost extending toward right. That's it for the rules.

Here's the next interesting detail: instead of actual nodes, heaps put values in arrays, using array indices to track parent-child relationships between values. [0] is held empty, and the root is at [1]. Hence, a value at [N] has children at [2N] and [2N+1], and its parent can be found at [N/2]. Tree traversal from an arbitrary node is quick.

Heap methods *top*, size, *isEmpty* and *contains* are trivial. For *push*, tree size will grow by one, and since our tree is complete, we know exactly where a "node" will be added. We don't know what value will go there, but let's start with the inserted value. From that spot, the value goes through a "promote" cycle: we successively compare it to its parent, swapping if its value is less. Once we can no longer promote the inserted value, insertion is complete.

MinHeap

Build (most of) a minHeap:push(value), top(), size(), isEmpty(), contains(value). Leave pop() for tomorrow.

Week 12 - Data Structures 2

CODING DOJO™

Overview

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 week 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? A *hash* function gives this data structure its quick-check, quick-retrieval feature, even when containing lots of data.

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

The word "associative" is used with associative arrays because they *associate* a certain key with a certain value. For example, if we use an associative array to track a specific user's attributes, we might have something like this: `{ name: "Marino", age: 27, IQ: 144, languages: ['Italian', 'English'], height: 181 }`. Here, we can 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. So, continuing our example and storing that one user's information in an array `["Marino",27,144,['Italian','English'],181]`, we could quickly access the user's age (27) or name ("Marino"), because we know the one and only one place in the array that we will always find the user's age (at index `[1]`) or name (at index `[0]`). We get the benefit of quick-retrieval only if 1) we decide, for each piece of information in our array, a specific index where we will always store it, and 2) we remember forevermore that decision (e.g. that `[0]` corresponds to name, `[1]` to age, etc). Can we make this more automatic, while still retaining quick-retrieval? A *hash function* can automatically pick indices for us.

Hash functions take an input (generally a string) and generate large, seemingly random (but repeatable) numbers, which we call *hash codes*. If we want a unique index for each key, we can use its hash code as the index – this way each key has a specific, reproducible index in our underlying array where its value will be stored. Note: hash codes could be truly huge (or negative), so to fit into our array, we must limit them to a manageable range.

We'll solve this by constraining our array to a certain capacity, then "modulo"ing the hash code so that it fits into that range. So to store the key/value `{ name: "Marino" }` into our data structure, we first get the hash code of the key 'name', then **mod** that hash code to get an index that fits within the capacity of our array, and finally 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 either keys or values *or* having to remember which index corresponds to which key. It's a beautiful thing.

We now know all we need in order to build the associative array data structure, which we can also call the *unordered map*. We call this data structure a *map*, because a map is a collection of keys that map to values. We call it *unordered*, because (unlike a BST or a Queue) we do not know the order or sequence the elements. Just because we can hash a certain key to a specific index (or bucket) in our array, that doesn't tell us anything about the keys that are hashed to the previous or subsequent buckets.

Week 12 - Data Structures 2 - Monday

CODING DOJO™

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 will also create the method `isEmpty()`, which may suggest that we add a `numKeys` attribute to our hashMap constructor.

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.

Today's challenges use these reference definitions:

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

// We 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 32bit int
  }
  return hash;
}
// Just like % but also handles neg values:
//   var myIdx = mod(myHashCode, arrSize);
function mod(input, div)
{ return (input % div + div) % div; }
```

add(key, value)

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

Second-level challenge: what if two keys hash to the same index, causing a "hash collision"? A small sllist or array may work better than simply storing a value.

isEmpty()

Create a method returning whether the map is empty. This is a one-liner, but requires changes elsewhere.

find(key)

Create a hashMap method that returns whether the tree contains the given key. If it does, return the associated value, otherwise return null.

Second-level challenge: if you altered `add()` to handle hash collisions, then you also need to extend `find()` to work with these alterations.

Tomorrow: Growing / shrinking the map

Week 12 - Data Structures 2 - Monday recap

CODING DOJO™

This week's BST challenges are based on the following reference definitions:

```
function hashMap(size) {
  this.capacity = size;
  this.table = [];
  this.numElements = 0;
}

// Just like % but also handles neg values:
//   var myIdx = mod(myHashCode, arrSize);
function mod(input, div) {
  return (input % div + div) % div;
}
```

```
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 32bit int
  }
  return hash;
}
```

add(key, value)

Create a method on the hashMap object to add a new key and value to the map. This entails hashing the key, mod'ing it into the size of your array, and placing the value there.
Also, if two keys hash to the same index, then we need a sllist or array to store *both keys and values*.

```
this.add = function(key, value) {
  if (key === undefined) { return false; }
  key += "";
  var hash = key.hashCode();
  var hashKey = mod(hash, this.capacity);
  if (this.table[hashKey] === undefined)
  { this.table[hashKey] = []; }

  //there's an array of keys at this bucket
  var keyArr = this.table[hashKey];
  for(var idx=0; idx<keyArr.length; idx++){
    if (keyArr[idx][0] === key) {
      keyArr[idx] = [key, value];
      return true; //replace existing value
    }
  }
  // if key not found, add the new pair
  keyArr.push( [key, value] );
  this.numElements++; //added for isEmpty()
  return true;
}
```

find(key)

Create a hashMap method that returns whether the tree contains the given key. If it does, return the associated value, otherwise return null.
Also, if we altered `add()` to handle hash collisions, we need to extend **find()** to work with these alterations.

```
this.find = function(key) {
  if (key === undefined) { return; }
  key += "";
  var hash = key.hashCode();
  var hashKey = mod(hash, this.capacity);
  if (this.table[hashKey] === undefined)
  { this.table[hashKey] = []; }

 //there's an array of keys at this bucket
  var keyArr = this.table[hashKey];
  for(var idx=0; idx<keyArr.length; idx++){
    if (keyArr[idx][0] === key)
    { return keyArr[idx][1]; }
  }  // returns 'undefined' if not found.
}
```

isEmpty()

```
// Easy peezy! (we also need to add
// "this.numElements = 0" to constructor)
this.isEmpty = function() {
  return (this.numElements == 0);
}
```

Week 12 - Data Structures 2 - Tuesday

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

remove(key)

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

loadFactor()

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

grow()

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

setSize(newSize)

Write methods to change the capacity of the internal array. If you change the array's size, you must rehash all keys, since the **mod** factor changed....

Tomorrow: pruning the tree

Week 12 - Data Structures 2 - Wednesday recap

CODING DOJO™

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

`this.find(key)`: create `results[]`, `if (multi)` then push val for (multiple) keys found, return results.

`this.remove(key)`: create `results[]`, `if (multi)` then splice (multiple) key/val found, return results.

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

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 key 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)` instead of `add([key, val])`.

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

Summary of Unordered Map/Set Data Structures:

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

unordered_multiset (from *unordered_set*)

constructor:

```
this.multi = allowDupes || false;
```

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

`this.find(key)`: use `numFound`: `if (multi)` then `numFound++` (multiple) found, return `numFound`.

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

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

Week 12 - Hashes & Dictionaries - Thursday

CODING DOJO™

By now you have seen how a data structure's interface (unordered map) can be completely decoupled from the underlying implementation (hashTable of arrays or sLists). In the same way that we dived into *unordered* sets and maps, you could now also dive into *ordered* sets and maps. Specifically, ordered data structures care about the order of values, so we could implement them with BSTs or heaps.

Today's algorithm challenges are less about coding and more about communicating ideas. How would you answer the following questions if you were 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?
Which queue methods are fast? (push, top/pop, min/max, contains, nextVal/prevVal, size)
Ditto - which methods 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?

What is a tree? Aren't 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 value to a tree?
How do I remove a value from a tree?
How would I check whether a tree is valid?
Which tree methods are fast? (add, remove, min/removeMin, max/removeMax, contains, nextVal/prevVal, size)
Ditto - which of these methods are slow?
Is there such thing as an 'unbalanced' tree?
Is there such thing as a 'full' tree?

What is a heap? How do heaps work, generally?
What are the advantages of using a heap?
What's the process for adding a value to a heap?
What's the process to remove a heap value?
How would I check whether a minheap is valid?
Which heap methods are fast? (push, pop, min/removeMin, max/removeMax, contains, nextVal/prevVal, size)
Ditto - which of these methods are slow?
Is there such thing as an 'unbalanced' heap?
Is there such thing as a 'full' heap?

What are unordered data structures? Is there more than one type? Describe. Why would I use each?
How are unordered maps represented in code?
What's the process for adding values to sets?
How do I remove a value from a set?
Is there a way to check validity of unordered sets?
Which unordered set methods are fast? (add, remove, min/removeMin, max/removeMax, contains, nextVal/prevVal, size)
Ditto - which of these methods are slow, for unordered sets?
Is there such thing as a 'full' set?
Is there an 'unbalanced' set?

What are ordered data structures? Is there more than one type? Describe. 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?
Is there a way to check validity of an ordered set?
Which ordered set methods are fast? (add, remove, min/removeMin, max/removeMax, contains, nextVal/prevVal, size)
Ditto - which methods are slow, for ordered sets?
Is there such thing as a 'full' ordered set?
Is there an 'unbalanced' ordered set?

What is the difference between sList and dList?
When would I use one versus the other? Question vs del
How are these implemented?
Is there a way to 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?

Tomorrow: putting your skills to the test

Week 12 - Hashes & Dictionaries - Thursday recap

CODING DOJO™

How would you briefly answer the following if asked in an interview?

What is a queue? When would I use one? Queue is a sequential data structure of variable size, used when data need to exit the data structure in the same order that they arrived.
What's the best way to implement a queue? Singly linked list (plus size) is most commonly used.
Can I implement this differently? Why? Circular queues use fixed-length arrays and front/back indices. We would use this in order to avoid constantly allocating/deallocating node objects.
What is a stack? When would I use one? Stack is a sequential data structure of variable size, used when data need to exit the data structure in the reverse order of their arrival.
What's the best way to implement a stack? Array is most commonly used.
Is there such thing as a hybrid queue/stack? Yes, a double-ended queue (deque: "deck") contains all the elements of a stack as well as those of a queue. This is best implemented as a doubly linked list.
Which queue methods are fast? (push, top/pop, min/max, contains, nextVal/prevVal, size) If implemented with singly linked list, then *push*, *top*, *pop*, *size* methods are O(1) for runtime.
Ditto - which methods are slow? How slow? If implemented with a singly linked list, the following methods are slow: *min*, *max*, *contains*, *prevVal*. For an sList, these all have O(n) runtime complexity.
If you needed queue.prevVal to be fast, would you change your underlying implementation? You could change underlying implementation to either doubly linked list (dList) or circular queue. If you won't need to rearrange the order of elements, and if size of queue is relatively bounded and constant, then circular queue would be an excellent choice. Otherwise go with a dList.
Is there such thing as an 'unbalanced' queue? No, you are just trying to confuse me.
Is there such thing as a 'full' queue? Yes: for *circular* queues that use fixed-length arrays.

What is a tree? Aren't they too complex to use? You obviously are a technophobe and need me to set you straight. These are not hard to use. A *binary* tree is a collection of nodes, starting from a single root node, where each node can have as many as two children, with no loops among the nodes.
Is there more than one kind of tree? Describe. If nodes can have more than two children, then they could be ternary trees or n-way trees. But what I think you mean is that a binary tree has no guarantee of any ordering of nodes, whereas a binary *search* tree enforces a specific way of ordering the nodes.
How are trees represented in code? Tree nodes are objects, each containing (at least) value and pointers to left and right children. A minimal tree contains the pointer to the root, which is a tree node.
What's the process for adding a value to a tree? Traverse the tree, staying left of greater values, and right of less-than-or-equal values; add a new node at the first unoccupied spot you find.
How do I remove a value from a tree? Traverse the tree as described above, searching for the value. If found, and if node has less than two children, directly connect node's parent to node's child. Otherwise, from there find next-larger value, and swap value into the node, removing next-larger node.
How would I check whether a tree is valid? Ensure that for each node, every value in the subtrees underneath are either less than (if in left subtree) or greater than or equal to (if in right subtree) that node's value. If the tree nodes also have a parent pointer, then ensure that the root's parent is null, and that each other node has a correct parent pointer referencing the node that in turn points to it.
Which methods are fast? (add/remove, min/removeMin, max/removeMax, contains, nextVal/prevVal, size) For a binary search tree, the following methods are (relatively) fast: *min*,

removeMin, *max*, *removeMax* and *contains*, *nextVal* and *prevVal* are O(logn) if we keep the tree relatively balanced. *Size* is O(1) as we cache this.
Ditto - which of these tree methods are slow? For a binary search tree, the following are (relatively) slow: *add* and *remove* are O(logn) at best -- slower if we self-balance during add and remove functions.
Is there such thing as an 'unbalanced' tree? Yes, a tree in which any node has left and right subtrees with heights that vary by more than one would be considered unbalanced.
Is there such thing as a 'full' tree? Yes, in full trees all nodes (except "leaf" nodes) have two children.

What is a heap? How do heaps work, generally? A minheap manages values by similar organization to a complete binary tree, where each node's value is less than or equal to values of both child nodes.
What are the advantages of using a heap? Rapid min-value retrieval without the slow add/remove.
What's the process for pushing a value to a heap? Add a node to our complete binary tree's next empty spot; "promote" that value upward (swapping values with parent) until heap properties are met.
What's the process to pop a heap value? Remove top node; swap value of the last node into top node; "demote" that value downward (swap values with upper child) until heap properties are met.
How would I check whether a minheap is valid? No parent node's value can exceed it's children's.
Which methods are fast? (push/pop, min/removeMin, max/removeMax, contains, nextVal/prevVal, size) If implemented as an array, the following methods are (relatively) fast: *push* and *pop* (which is *removeMin*) are O(logn) or better. *Min* and *size* are O(1).
Ditto - which of these heap methods are slow? If implemented as an array, the following methods are (relatively) slow: *max*, *removeMax*, *contains*, *nextVal* and *prevVal* are all O(n).
Is there such thing as an 'unbalanced' heap? No, you can't confuse me. They are always balanced.
Is there such thing as a 'full' heap? No, you are confused. They are always full (at least complete).

What are unordered data structures? Is there more than one type? Describe. *Unordered* data structures store information without regard to how values or arrival times interrelate. These include *maps*, *sets*, *multimaps* and *multisets*. Maps/multimaps store keys and values; sets/multisets store keys. Multimaps/multisets can contain a key multiple times (in a multimap, perhaps with different values).
Why would I use each? If quantity does not matter, but only membership, use sets or maps instead of multisets or multimaps. If only direct data matters, and not additional data (if only the key is needed, and not an additional value), then use sets or multisets, instead of maps or multimaps.
How are unordered maps represented in code? Unordered maps are commonly implemented as hashtables, each bucket containing a singly linked list of key-value records.
What's the process for adding values to sets? Hash the key, mod the hash value to fit into the number of buckets, and add the actual key to the end of the sList found at that bucket.
How do I remove a value from a set? Hash the key, mod the hash value to fit into the number of buckets, and remove the actual key from the sList found at that bucket.
Is there a way to check validity of unordered sets? Ensure buckets contain keys that hash there.
Which unordered set methods are fast? (add, remove, min/removeMin, max/removeMax, contains, nextVal/prevVal, size) In a hashtable of sLists where the loadFactor is kept low, the following methods are (relatively) fast: *add*, *remove* and *size* are approx. O(1).
Ditto - which of these methods are slow, for unordered sets? In a hashtable of sLists where loadFactor is kept low, these are slow: *min*, *removeMin*, *max*, *removeMax*, *contains*, *nextVal*, *prevVal*.
Is there such thing as a 'full' set? No, you are just trying to confuse me.
Is there an 'unbalanced' set? No, you are just trying to confuse me. What is it with you guys?

Next week: moving forward to another topic, bit by bit....

What are ordered data structures? Is there more than one type? Describe. Why would I use each? *Ordered* data structures store information as well as how values or arrival times interrelate.
How are ordered maps represented in code? Ordered data structures are represented as BSTs.
How do I add values to an ordered set? See the above explanation for how to add a value to a BST.
How do I remove a value from an ordered set? See above explanation for removeing a BST value.
Is there a way to check validity of an ordered set? See above explanation for checking BST validity.
Which ordered set methods are fast? (add, remove, min/removeMin, max/removeMax, contains, nextVal/prevVal, size) See prev (min/removeMin, max/removeMax, contains, nextVal, prevVal, size).
Ditto - which methods are slow, for ordered sets? See previous answers for BST (add and remove).
Is there such thing as 'full' ordered sets? No, not really.
Is there an 'unbalanced' ordered set? No, not really.

What is the difference between sList and dList? sList (singly linked list) and dList (doubly linked list) differ only in that a dList has nodes that contain a prev pointer (in addition to the next pointer in sLists).
When would I use one versus the other? Use a dList if you need to traverse backwards across the list, even if moving only one node backward frequently, or if moving significantly backward infrequently.
How are these implemented? dList nodes have prev pointers. Sometimes head.prev points to tail.
Is there a way to check whether a dList is valid? Does every node.prev.next point back to self?
Which list methods are significantly different, between sLists and dLists? (front, back, pushFront/popFront, pushBack/popBack, min/max, contains, nextVal/prevVal, size) Of the methods given, between sLists and dLists only these methods significantly differ: *popBack* and *prevVal*.
Is there such thing as an 'unbalanced' dList? No, this doesn't make sense, people.
Is there such thing as a 'full' dList? No, no, a thousand times no.

Next week: moving forward to another topic, bit by bit....

Week 12 - Hashes & Dictionaries - Friday

Today you will take the belt exam in Algorithms, leading to the rare and much-coveted Blue Belt.

Best of luck, and remember all our super suggestions, good guidance and miscellaneous pearls of wisdom!

Next week: moving forward to another topic, bit by bit....