**ALGORITHM:**

1. **Hash table**

\_Putting in an item into a table using keys and hash algorithm

\_Have constant O(1) lookup time b/c it uses keys

\_No concept of order

\_Collision

* are **unordered** (the keys are not guaranteed to stay in the same order)
* can use **many types of objects as keys** (commonly strings)

Q: Hash vs Array?

A: Arras are fixed in size. Hash tables have no size limit

\_Insertion in hash is O(1). Array it’s O(n)

\_

**II- Reduce: Reduce all the numbers to 1 number (like average)”**

\_Arr.reduce(callback{currentValue, initialValue})

Ex:

var arr = [0,1,2,3]

var x = arr.reduce(function(initial,total){

return initial + total}, 3);

* Reduce takes in an initial value, a total (something that adds together), and then return the two, as well as an additional number to plus together

Ex:

Q: /\* Popular Ice Cream Totals Quiz

\*

\* Using the data array and .reduce():

\* - Return an object where each property is the name of an ice cream flavor

\* and each value is an integer that's the total count of that flavor

\* - Store the returned data in a new iceCreamTotals variable

\*

\* Notes:

\* - Do not delete the data variable

\* - Do not alter any of the data content

\*/

A:

const iceCreamTotals = data.reduce((acc, person) => {

person.favoriteIceCreams.forEach(iceCream => {

if(acc[iceCream]){

acc[iceCream]++

} else{

count = 1;

acc[iceCream] = count;

}

});

return acc

}, {});

//the starting could be anything. In this case it’s an object. The end (acc) has to be an object too. If it’s anything else, the function will stop working on its second iteration. Acc here through each iteration is the act of adding things to the starting object

// if(acc[iceCream]) is to check whether there exists an iceCream flavor already

// acc[iceCream] is also used to access the value of the key iceCream

// data.reduce((acc, person): the second argument is the current Value. Current value here means which item in the array it’s currently at.

//so I do person.favoriteIceCreams because I’m trying to get inside each person’s favorite ice cream

\_For loop: Separate by ; not ,

## \_The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

do {  
*code block to be executed*}  
while (*condition*);

\_What Objects lok like:

var car = {type:"Fiat", model:"500", color:"white"};

Q: To replace an item in an array, use splice:

A: Ex:

var arr =[1,2,3]

arr.splice(0,1,2)

// arr = [2,2,3]

//splice(start,length, what to replace with)

\_FOR/IN LOOP:

var text = "";  
var x;  
for (x in person) {  
    text += person[x];  
}

CANNOT use “return” in LOOPs

\_FOR/OF loop:

**const** iterable = ['a', 'b'];

**for** (**const** x **of** iterable) {

console.log(x);

}

//Objects not iterable, so can’t use For..Ops

Q: SAMPLE TEST:

/\*

\* anagramPalindrome

\*

\* Write a function which accepts an input word and returns true or false if there exists

\* some anagram (permutation) of that input word that is a palindrome.

\*

\* "cat" => "tac", "tca"

\*/

A:

//HIGH LEVEL:

BASICALLY PUT EACH LETTER OF THE WORD INTO A HASH TABLE.

HASH TABLE: ALPHABETICAL ORDER: VALUE

SO HASH TABLE IS BASICALLY AN {}

SO YOU STORE HOW MANY TIMES EACH LETTER APPEARS AS A KEY

//SO SINCE A PALINDROME IS EITHER IN THIS FORM: ABA or AA. But remember this is a permutation so it’s not in that order

//SO STRATEGY IS, IF THERE’S ONLY ONE LETTER, AND EVERYTHING ELSE’S VALUE IS DIVIDED BY 2, THEN THAT’S A PALINDROM! (ABA case)

//ALSO, OF COURSE, IF EVERTHING’S VALUE IS DIVIDED BY 2 (AA case) then it’s a palindrom

// else, nope

var anagramPalindrome = function(word){

let frequencyOfLetters={};

for(var i =0; i<word.length; i++){

let count = 0;

//So that you can reset count to 0 every time

//and why you need to? Because you want to increment the value count, not the count itself by more than 1

if(frequencyOfLetters[word[i]]){

// don ‘t do frequencyOfLetters.word because it’ll look for the literal string ‘word’

frequencyOfLetters[word[i]] ++;

}

else{

count =1;

frequencyOfLetters[word[i]]= count

}

}

let check = true;

let frequencyCount = 0;

//outside so that value of count is tallied up each time and not set to 0

for(let letter in frequencyOfLetters){

//if values of all keys is divided by 2, return true

//+) if only one key is false, also return true

//if values of all but one key is divided by 2, return true

//otherwise, false

if ((frequencyOfLetters[letter] %2) ===1 ){

frequencyCount ++;

}

if ((frequencyOfLetters[letter] %2) ===0 || frequencyCount ===1) {

check;

}

else{

check= false;

}

}

return check;

};

console.log(anagramPalindrome("carrace")); // true

console.log(anagramPalindrome("cutoo")); // false

console.log(anagramPalindrome("an")); // false

console.log(anagramPalindrome("dddaaaayyyy")); // true

console.log(anagramPalindrome("anna")); //true

console.log(anagramPalindrome("bbd")); //true

**BIG O: Worst case scenario**

\_Time complexity: measure the amount of computations

\_Space complexity: measures the amount of memory

//measuring against the size of input

Q: What is scaling?

// Take in an integer and print its value

function printInteger(num) {

console.log(num);

}

**A:** If your answer was the **value** of num then you are correct. We would be scaling the value of num itself.

Lets try another problem:  
**Q:** For the following code below, what is scaling?

// Print the first item in the array

function printFirst(arr) {

console.log(arr[0]);

}

A: In this case, its the **length of the array**. If you are given an array as input, it usually is the length of the array.

| **Input** | **Common Factor that is Scaling** |
| --- | --- |
| Integer | Magnitude of number |
| String | Length of String |
| Array | Length of Array |
| LinkedList | Number of nodes |
| Tree | Number of nodes |
| Hashtable | Number of key-value pairs |
| Matrix | Width and height of matrix |
| Graph | Number of vertices and edges |

**One Unit of Time**

* A unit of time can be one arithmetic operation: 5 + 7
* Or printing something: console.log('hello')
* Or instantiating a new variable: let name = 'foo';
* Or accessing an item in an Array: arr[5]
* Or returning something: return 'foobar';

**One Unit of Space**

* A unit of space can be creating a single new variable: var i = 1;
* Or creating an empty array: var list = [];
* Or adding a new item to your list: list.push('foo');
* Or adding a key-value pair into a hashtable: obj[foo] = 'bar';

Q: How many operations does this function have? Time Complexity?

A:

function printFirstLast(arr) {

console.log(arr[0]); // 2

console.log(arr[arr.length-1]); // 4

}

//6

// But O(1): Why? Because we drop the coefficient 6. And also because O(1) means every time we do the same operations, it’s constant time

Google Big O quiz

Q: If you have 2 for loops in a function, what is its time complexity?

A: O(n) because you drop the constant (2)

Q: For loop’s time complexity?

A: Linear—O(N)

Q: While loop?-- tricky

A: Whenever it has to do with .length, it’s going to be O(N), even if it doesn’t start from 0, because you’ll simplify the operations anyways.

Q: Best way to analyze big O?

A: By chunks & use magnitude:

// given an array of integers, return all the even items.

function evens(arr){

let results = []; // constant

for(let i = 0; i < arr.length; i++) { // linear

if(arr[i] % 2 === 0) { // constant

results.push(arr[i]);

}

return result; // constant

}

//It’s linear (arr.length), and everything inside is constant, so it’s O(N)

Q: Nested vs UnNested for loops?

A: Nested – O(N^2) 🡪 The inner loop isn’t exactly O(N), but because we drop the constant, so it’s O(N).

UnNested – O(N)

Q: Time Complexity of: make flash card with these

A:

Array - indexOf

Array - map

Array – forEach – O(n)

Array - reverse

Array – sort

Array-push: O(1)

Array- pop O(1)

Shift – O(N)

Splice—O(N) – it has to find

Sort – O(N log N)

Linked List – O(1)

Q: For DATA STRUCTURE METHODS, what case should we pay attention to?

A: Average case

|  |  |  |
| --- | --- | --- |
| Algorithm | Time Complexity | |
|  | Average |
| [Quicksort](http://en.wikipedia.org/wiki/Quicksort) | Θ(n log(n)) |
| [Mergesort](http://en.wikipedia.org/wiki/Merge_sort) | Θ(n log(n)) |
| [Timsort](http://en.wikipedia.org/wiki/Timsort) | Θ(n log(n)) |
| [Heapsort](http://en.wikipedia.org/wiki/Heapsort) | Θ(n log(n)) |
| [Bubble Sort](http://en.wikipedia.org/wiki/Bubble_sort) | Θ(n^2) |
| [Insertion Sort](http://en.wikipedia.org/wiki/Insertion_sort) | Θ(n^2) |
| [Selection Sort](http://en.wikipedia.org/wiki/Selection_sort) | Θ(n^2) |
| [Tree Sort](https://en.wikipedia.org/wiki/Tree_sort) | Θ(n log(n)) |
| [Shell Sort](http://en.wikipedia.org/wiki/Shellsort) | Θ(n(log(n))^2) |
| [Bucket Sort](http://en.wikipedia.org/wiki/Bucket_sort) | Θ(n+k) |
| [Radix Sort](http://en.wikipedia.org/wiki/Radix_sort) | Θ(nk) |
| [Counting Sort](https://en.wikipedia.org/wiki/Counting_sort) | Θ(n+k) |
| [Cubesort](https://en.wikipedia.org/wiki/Cubesort) | Θ(n log(n)) |

**Common Data Structure Operations**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Data Structure | Time Complexity | | | | | |  |
|  | Average | | | | | |
|  | Access | Search | | Insertion | | Deletion |
| [Array](http://en.wikipedia.org/wiki/Array_data_structure) | Θ(1) | Θ(n) | Θ(n) | | Θ(n) | |
| [Stack](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)) | Θ(n) | Θ(n) | Θ(1) | | Θ(1) | |
| [Queue](http://en.wikipedia.org/wiki/Queue_(abstract_data_type)) | Θ(n) | Θ(n) | Θ(1) | | Θ(1) | |
| [Singly-Linked List](http://en.wikipedia.org/wiki/Singly_linked_list#Singly_linked_lists) | Θ(n) | Θ(n) | Θ(1) | | Θ(1) | |
| [Doubly-Linked List](http://en.wikipedia.org/wiki/Doubly_linked_list) | Θ(n) | Θ(n) | Θ(1) | | Θ(1) | |
| [Skip List](http://en.wikipedia.org/wiki/Skip_list) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | | Θ(log(n)) | |
| [Hash Table](http://en.wikipedia.org/wiki/Hash_table) | N/A | Θ(1) | Θ(1) | | Θ(1) | |
| [Binary Search Tree](http://en.wikipedia.org/wiki/Binary_search_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | | Θ(log(n)) | |
| [Cartesian Tree](https://en.wikipedia.org/wiki/Cartesian_tree) | N/A | Θ(log(n)) | Θ(log(n)) | | Θ(log(n)) | |
| [B-Tree](http://en.wikipedia.org/wiki/B_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | | Θ(log(n)) | |
| [Red-Black Tree](http://en.wikipedia.org/wiki/Red-black_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | | Θ(log(n)) | |
| [Splay Tree](https://en.wikipedia.org/wiki/Splay_tree) | N/A | Θ(log(n)) | Θ(log(n)) | | Θ(log(n)) | |
| [AVL Tree](http://en.wikipedia.org/wiki/AVL_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | | Θ(log(n)) | |
| [KD Tree](http://en.wikipedia.org/wiki/K-d_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | | Θ(log(n)) | |

**MAGNITUDE:**

| **Input** | **Common Examples** |
| --- | --- |
| Constant: O(1) | Array access, arithmetic operators |
| Logarithmic: O(log(N)) | Binary search, binary search tree retrieval/insertion (balanced) |
| Linear: O(N) | Looping through an array or hashtable |
| Quasilinear: O(Nlog(N)) | Quicksort, mergesort, heapsort |
| Quadratic: O(N^2) | Nested loops |
| Polynomial: O(N^C) | Deeply nested loops |
| Exponential: O(C^N) | Multiple-recursion |
| Factorial: O(!N) | Permutations |

If you see nested loops, the run time will be O(n^level of nesting)

Ex: A function with a loop inside a loop inside a loop: O(n^3)

Q: Big O usually is referred to?

A: Time complexity

Q: Space complexity?

A: How much additional memory does the algorithm require beyond what needs to be allocated for the input themselves?

Q: space complexity of this?

A: **function** **total**(array) {

**var** total = 0;

**for** (**var** i = 0; i < array.length; i++) {

total += array[i];

}

**return** total;

}

The space complexity is just O(1), since we only require one additional unit of space, for the number stored in total.

Q: space complexity of this?

**function** **double**(array) {

**var** newArray = [];

**for** (**var** i = 0; i < array.length; i++) {

newArray.push(2 \* array[i]);

}

**return** newArray;

}

// Here, our function double takes each element of the input array, doubles it, and returns a new array of doubled values. In this case, both the time and space complexities are O(n). Space complexity is larger in this case because we need n additional units of space: one for each element in the original array.

Q: For loop is

A: O(n)

It runs slower the more things you give it, but that should grow at a predictable rate.

Q: O complexity of this?

A:

function printAllItemsTwice(theArray) {

theArray.forEach(function(item) {

console.log(item);

});

// once more, with feeling

theArray.forEach(function(item) {

console.log(item);

});

}

// O(2n) => O(n)

Q: This?

A:

function printAllNumbersThenAllPairSums(arrayOfNumbers) {

console.log('these are the numbers:');

arrayOfNumbers.forEach(function(number) {

console.log(number);

});

console.log('and these are their sums:');

arrayOfNumbers.forEach(function(firstNumber) {

arrayOfNumbers.forEach(function(secondNumber) {

console.log(firstNumber + secondNumber);

});

});

}

O(n^2). Even though it’s O(n+ n^2)

Q: Space complexity of this?

A:

function sayHiNTimes(n) {

for (var i = 0; i < n; i++) {

console.log('hi');

}

}

// O(1): since we don’t allocate any new **variable**

This function takes *O*(*n*) space (the size of hiArray scales with the size of the input):

function arrayOfHiNTimes(n) {

var hiArray = [];

for (var i = 0; i < n; i++) {

hiArray[i] = 'hi';

}

return hiArray;

}

Space complexity = additional space, not including the space taken already by the input

Space complexity seems to only have O(1) or O(n) value

Q: let vs constant. Which one is stricter?

A: const

**FREQUENCY COUNTING**

Q: when to use Hashtable, Array or Set?

A:

Hashtable: General all purpose use

Array: Values in the collection that are of a **small range of integer values.**

Set: If only needed to track if something exists.

**HASH TABLE:**

\_See Algorithm practice

\_Binary search’s O time: O(log N)

**BINARY SEARCH:**

\_Basically divide and conquer. Split down the middle

**RECURSION:**

\_Always have a base case

\_Each iteration is modified so that it eventually reach the base case

ex: function sumRange(num){

if(num === 1) return 1;

return num + sumRange(num-1);

}

//Recursion is 10 times slower than loop!

If I want a function to be called multiple times in an outer function, use a helper function, or pure recursion.

Ex:

Q: To help with scope in recursion, we can create a wrapper or helper function which will be called multiple times in an outer function (to provide additional scope). This is done through a process called helper method recursion. Let's start by writing a function called all which accepts an array and a callback and returns true if every value in the array returns true when passed as parameter to the callback function.

A:

function allRecursive(array, condition) {

var copy = array.slice();

function allRecursiveHelper(arr, cb){

if (arr.length === 0) return true;

if (condition(arr[0])){

arr.shift();

return allRecursive(arr,condition);

} else {

return false;

}

}

return allRecursiveHelper(copy, condition);

}

var numbersArray = [1,2,3,4,5];

allRecursive(numbersArray, function(v) {

return v > 0;

});

Or I can do it straight with pure recursion:

function allRecursive(array, condition) {

var copy = copy || array.slice();

if (copy.length === 0) return true;

if (condition(copy[0])){

copy.shift();

return allRecursive(copy,condition);

} else {

return false;

}

}

var numbersArray = [1,2,3,4,5];

allRecursive(numbersArray, function(v) {

return v > 0;

});

**STACK**: Last in first out!!

**SORTING: BUBBLE, INSERTION SORT**

1. **BUBBLE SORT:**

\_Most basic sort: sequentially go through array and compare 2 values at a time, swapping them if necessary. Repeats until no swaps are required

\_Runtime: O(n^2)

Can write it with 2 nested While loops/ For loops, or do-while with a nested For loop

Ex:

function bubble(arr) {

var len = arr.length;

for (var i = 0; i < len ; i++) {

for(var j = 0 ; j < len - i - 1; j++){ // this was missing

if (arr[j] > arr[j + 1]) {

// swap

var temp = arr[j];

arr[j] = arr[j+1];

arr[j + 1] = temp;

}

}

}

return arr;

}

document.write(bubble([1,9,2,3,7,6,4,5,5]));

Declare function:

function myFunction(p1, p2) {

return p1 \* p2; // The function returns the product of p1 and p2

}

// no “=”

ex:

Q: // Write a function called power which takes in a base and

// an exponent. If the exponent is 0, return 1.

// Otherwise, return the result of the base multiplied by the

// power function to the exponent - 1. You can think of it

// in terms of this example:

// 2^4 = 2 \* 2^3;

// 2^3 = 2 \* 2^2;

// 2^2 = 2 \* 2^1;

// 2^1 = 2 \* 2^0; // once our exponent is 0 we

// KNOW that the value is always 1!

A: function power(base, exponent) {

if (exponent ===0 ){

return 1

}

return base \* power(base,exponent -1)

}

power(2,1);

power(2,2);

power(2,3);

power(2,4);

Ex 2:

Q: Write a function that returns the factorial

// of a number. As a quick refresher, a factorial of a number

//is

// the result of that number multiplied by the number before it,

// and the number before that number, and so on,

// until you reach 1. The factorial of 1 is just 1.

// For example:

// factorial(5); // 5 \* 4 \* 3 \* 2 \* 1 === 120

A: function factorial(number){

if(number ===1){

return 1

}

return number \* factorial(number-1);

}

console.log(factorial(5))

1. **Insertion sort:**

\_Runtime: O(n^2).

\_ Still not clear on what it is

\_But 1.5 faster than bubble sort

Ex: A very good explanation: Look at it again first thing tmr—1pomodoro on it

function insertionSort (items) {

for (var i = 0; i < items.length; i++) {

let value = items[i]

// store the current item value so it can be placed right

for (var j = i - 1; j > -1 && items[j] > value; j--) {

// loop through the items in the sorted array (the items from the current to the beginning)

// copy each item to the next one

// notice j+1 here, since j is decreasing, index is actually j-1

items[j + 1] = items[j]

}

// the last item (beginning of array) we've reached should now hold the value of the currently sorted item

items[j + 1] = value

}

//what is this ‘list’ thing?

return list

}

const list = [54, 26, 93, 17, 77, 31, 44, 55, 20]

console.log(insertionSort(list)) // [ 17, 20, 26, 31, 44, 54, 55, 77, 93 ]// skip the first item because we already consider it in the sorted portion of the array

//Come bacm and leaern it tmr

1. **Quick sort:**

\_ Pick a book arbitrary and partition books that come before it to the left, and books that come after it to the right

\_ Do the same thing with each halves and then sort them using another method

\_O(nlog n): Most efficient sort

Q: Sum of 1+2 + … (n-1)? (n= length of lsit)

A: n(n-1) /2

\_In codepad, have to use console.log()

Q: is this code bubble sort or insertion sort?

function insertionSort(ul){

    for(var i =0; i< ul.length; i++){

        for(var j = i-1; j<ul.length; j++){

            if(ul[j] > ul[j+1]){

            [ul[j],ul[j+1]] = [ul[j+1], ul[j]]

            }

        }

    }

    return ul

}

var ul = [5, 3, 1, 2, 4];

console.log(insertionSort(ul));

A: bubble sort, cuz we don’t swap items in insertion sort

// whenever I have 2 loops, where index of 1 loop is behind the other, then the inner loop is for 1 iteration, whereas the outer loop is for all iterations

**SET**

Q: What’s Set?

A: an object of unique items. Like “hash”, but without values

Q: how to create a set?

A: 1) const games = new Set();

console.log(games);

// Set {}

ex: const games = new Set(['Super Mario Bros.', 'Banjo-Kazooie', 'Mario Kart', 'Super Mario Bros.']);

console.log(games);

// Set {'Super Mario Bros.', 'Banjo-Kazooie', 'Mario Kart'}

* So hash is just an object, but Set I have to create new Set()

Q: How do modify set?

A: .add() or .delete() or .clear() to delete every item

\_ .delete returns true or false depending on successful deletion

Ex:

const games = new Set(['Super Mario Bros.', 'Banjo-Kazooie', 'Mario Kart', 'Super Mario Bros.']);

games.add('Banjo-Tooie');

games.add('Age of Empires');

games.delete('Super Mario Bros.');

Ex:

games.clear()

console.log(games);

Q: To check the size of a Set

A: set.size()

console.log(months.size);

Q: Checking if an item exists:

A: set.has(item)

Q: set.values()?

A: return values in a set

Q: To move through values in a set?

A: .next()

Ex:

const iterator = months.values();

iterator.next();

Q: How to use “for… of?”

A: It’s like for in, but for array and set

Do NOT use “for..of” when in a combination of array and set. Why? Because it runs “iterator” beneath it!