My Programming Notes

Data Structures

- Arrays
- Stacks
- Queues
- Linked Lists
- Trees
- Tries
- Graphs
- Hash Tables

Algorithms

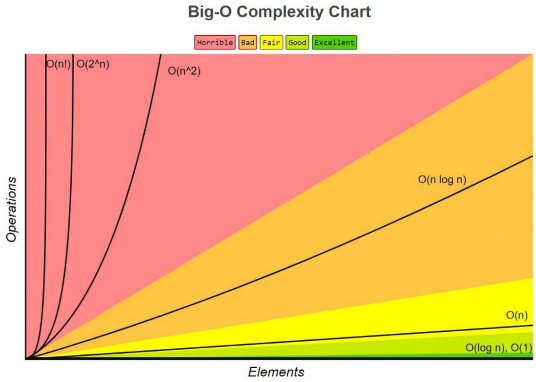
- Sorting
- Dynamic Programming
- BFS + DFS (Searching)
- Recursion

Big O-Notation

- Asymptotic analysis — how well a problem is solved and the performance of an algorithm at the limit.

What is good code?

- Readable — is your code clean? Can others understand your code?
- Scalable — when new requirements come to you, in how much of a change are you able to adapt to the new requirement.
  - The lesser the lines required, the more scalable the code is.



When we talk about Big-O and scalability of code, it simply means that when the input gets larger, how much does the algorithm or function slow down?

i.e. As the elements in the chart increases, how many more operations need to be done?

*How much does the algorithm slow down? *

Cheat Sheet (Big-O)

O(1) → Constant, no loops

O(log N) → Logarithmic, usually searching algorithms have log n if they are sorted. (Binary Search)

O(n) → Linear – for loops, while loops through n items

O(n log(n)) → Log linear – usually sorting operations

$O(n^2)$ → Quadratic – every element in a collection needs to be compared to every other element. Two nested loops

$O(2^n)$ → Exponential – recursive algorithms that solves a problem of size N

O(n!) → Factorial – you are adding a loop for every element.

Practical Code Examples of Big-O Notation

Constant time algorithm – O(1):
Logarithmic algorithm – O(log n):
Linear Time Algorithms – O(n):
N log N Time Algorithms – O(n log n):
Polynomial Time Algorithms – O(n^p):
Exponential Time Algorithms – O(k^n):

Factorial Time Algorithms – O(n!):

a = 1, print(a)

Note: It doesn't matter what the variable is or what is printed. It stays constant.

binary search algorithm

for loops & while loops

nested search (sorting algorithms)

nested loops

```
for (int i=1, i <= Math.pow(2, n); i++) {
        System.out.println("Hey — I'm busy looking at: " + i);
}
for (int i = 1; i <= factorial(n); i++) {
        System.out.println("Hey — I'm busy looking at: " + i);
}
```

This runs at 2^8 times
$2^8 = 256$ times

Example of using this IRL is the traveling salesman while using a brute-force approach to solve it.

```
1  // What is the Big O of the below function? (Hint, you may want to go line by line)
2  function funChallenge(input) {
3    let a = 10;     O(1)
4    a = 50 + 3;     O(1)
5
6    for (let i = 0; i < input.length; i++) {  O(n)
7      anotherFunction();    O(n)
8      let stranger = true;   O(n)
9      a++;   O(n)
10   }
11   return a;   O(1)
12 }
```

funChallenge

O(3 + n + n + n + n)

O(3 + 4n) ← The time complexity it took for the function to run

* This can get simplified to just O(n)

```
1  // What is the Big O of the below function? (Hint, you may want to go line by line)
2  function anotherFunChallenge(input) {
3    let a = 5;     O(1)
4    let b = 10;    O(1)
5    let c = 50;    O(1)
6    for (let i = 0; i < input; i++) {
7      let x = i + 1;     O(n)
8      let y = i + 2;     O(n)
9      let z = i + 3;     O(n)
10
11   }
12   for (let j = 0; j < input; j++) {
13     let p = j * 2;    O(n)
14     let q = j * 2;    O(n)
15   }
16   let whoAmI = "I don't know";  O(1)
17 }
```

anotherFunChallenge

O(4 + n + n + n + n + n)

O(4 + 5n) OR O(n)

Simplifying Big-O (Rule Book)

1.) Worst Case

- What is the worst case scenario?
- Big O establishes a worst case run time

```
1  //#1 -- For loop in Javascript.
2  const nemo = ['nemo'];
3
4  function findNemo1(array) {
5    for (let i = 0; i < array.length; i++) {
6      if (array[i] === 'nemo') {
7        console.log('Found NEMO!');
8      }
9    }
10 }
11
12 findNemo1(nemo);
```

When looking at this example, we can see that a simple search is being done here. Since "nemo" is the first and only item in the array, O(1) is the best case scenario. But what if there were 10, 100, or even 1000 items in the array? Since Big-O Notation focuses on the worst-case-scenario, the answer would be O(n) for a simple search like the example given.

- The worst-case-scenario gives us reassurance that the time complexity will never be slower than O(n).

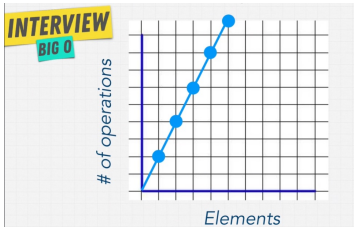
2.) Remove Constants

```
function printFirstItemThenFirstHalfThenSayHi100Times(items) {
    console.log(items[0]);                          O(1)

    var middleIndex = Math.floor(items.length / 2);
    var index = 0;

    while (index < middleIndex) {                   O(n/2)
        console.log(items[index]);
        index++;
    }

    for (var i = 0; i < 100; i++) {                 O(100)
        console.log('hi');
    }
}
```

When looking at the function on the left, the Big-O Notation is $O(1+n/2+100)$. Rule #2 tells us to drop the constants. Therefore, the time complexity for this function gets simplified to $O(n)$.

* In the situation where n is being divided by 2, we drop the division because $n/2$ has a decreasingly significant effect. *

The graph on the right is provided as proof that we are allowed to remove the constants because it doesn't matter how steep the line is. What we care about is how the line moves as our input increases.



3.) Different terms for inputs

When given the two functions below, can we say that the Big-O Notation for the right picture is still O(n)?

```
function compressBoxesTwice(boxes) {
    boxes.forEach(function(boxes) {
        console.log(boxes);
    });


    boxes.forEach(function(boxes) {
        console.log(boxes);
    });
}
```

VS

```
function compressBoxesTwice(boxes, boxes2) {
    boxes.forEach(function(boxes) {
        console.log(boxes);
    });


    boxes2.forEach(function(boxes) {
        console.log(boxes);
    });
}
```

The answer is: NO

The reason the answer is no is because of the different terms for different inputs. Therefore, both "boxes" and "boxes2" are both two different inputs. One input can be 100 items long while the other input is only 1 item long.

The big O-notation for the first function shown on the left is $O(n)$. The big o-notation for the second function shown on the right is $O(n+m)$.

* Just because you see two for-loops, one right after the other, it does not mean they're looping over the same items. *

- **Nested loops**

```
// Log all pairs of array
const boxes = ['a','b','c','d','e'];

function logAllPairsOfArray(array) {
  for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length; j++) {
      console.log(array[i], array[j])
    }
  }
}

logAllPairsOfArray(boxes)
```

Since there are nested for-loops within the function shown here, we multiply instead of adding like what we have done previously.

In this case, we can only see 2 nested for-loops.
$\therefore$ $O(n * n) = O(n^2)$ → worst case scenario

With $O(n^2)$ being extremely slow in speed, most interviews will ask for a solution to solve the issue w/ the speed.

4.) Drop Non Dominants

```
function printAllNumbersThenAllPairSums(numbers) {

  console.log('these are the numbers:');
  numbers.forEach(function(number) {
    console.log(number);
  });

  console.log('and these are their sums:');
  numbers.forEach(function(firstNumber) {
    numbers.forEach(function(secondNumber) {
      console.log(firstNumber + secondNumber);
    });
  });
}

printAllNumbersThenAllPairSums([1,2,3,4,5])
```

We can see from the example given, there is one for loop and a nested for loop. Based on what we see, we can say the Big O-Notation for this function is as follows:

$= O(n + n * n)$
$= O(n + n^2)$
$= O(n^2 + n)$
$= O(n^2)$

Since n^2 is the most dominant term, we are able to simplify it to just $O(n^2)$

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Both tables were found on https://www.bigocheatsheet.com

3 Pillars of Programming

1.) Readable - writing readable/clean code that others can maintain
2.) Memory - what's the memory usage of the code?
3.) Speed - time complexity, the code scales well

Space Complexity

When a program executes, it has two ways to remember things:

1.) Heap - where the variables that have values assigned to them are stored.
2.) Stack - where we are keeping track of our function calls.

Sometimes we want to optimize for using less memory instead of using less time.

What causes space complexity?

- Variables
- Data Structures
- Function Calls
- Allocation