Big O Notation

Information from: Cracking the Coding Interview, and HackerNoon- Big O for Beginners

****Not meant to be in depth but an overview of Big O**

In latence terms it is how we describe the efficiency of algorithms. There also exists **Big Omega** and **Big Theta.** But we rarely use that especially for coding interviews.

Big O

This is used to describe the upper bound in time, AKA the worst case scenario in the efficiency of the algorithm. Common representation: **O(1)**, **O(n)**, **$O(n^2)$**

O(1) - Constant Time

Our algorithm will always take the same amount of time no matter how many executions of the algorithm take place. Very predictable and scalable [example](#):

```
const items = [0, 1, 2, 3, 4, 5];

const logFirstTwoItems = (items) => {
  console.log(items[0]); // O(1)
  console.log(items[1]); // O(1)
}

logFirstTwoItems(items); // O(2)
```

O(n)- Linear Time

In a traditional for loop the time complexity is O(n) as it runs for every value in the for loop. The operations increase in a linear fashion according to the inputs. (n) represents the number of inputs.

O(n^2) - Quadratic Time

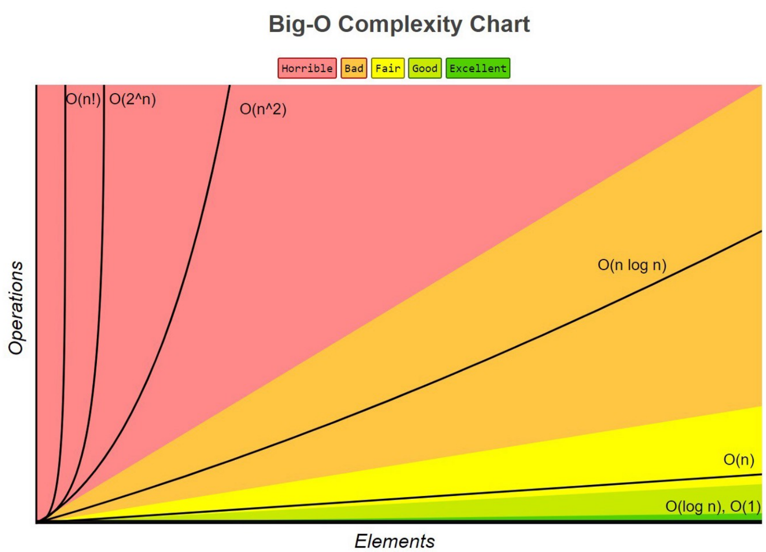
Say we have an algorithm that logs a series of pairs from an array of items.

```
const items = ['One', 'Two', 'Three', 'Four', 'Five'];

logAllPairs = (items) => {
  for (let i = 0; i < items.length; i++) {
    for (let j = 0; j < items.length; j++) {
      console.log(items[i], items[j]);
    }
  }
}

logAllPairs(items)
```

Now a good rule of thumb is if we see nested loops then we can use multiplication to get the notation. So above we have (n * n) or n^2. Known as quadratic time which means for every time our input increases we increase the operations done quadratically. We DO NOT want this.



Calculating Big O

You can go thru each line and and establish its individual runtime and put them into a simple math equation ie: $O(4 + 5n + n^2)$ so here we have a **constant + linear + quadratic.**

Now there are 4 rules that we can follow to simplify the above equation:

1. Assume the worst
2. Remove constants
3. Use different terms for inputs
4. Drop any non dominants

Let's start with rule 2/3, a constant is a number generally but lets only look at **5n**. Now we know that mathematically 5 * n does not really increase our time complexity too significantly so we can drop them as they wont ever change from **5**. And now we are left with $O(4 + n + n^2)$ 4 the more important one IMO. Now according to the above chart both **O(4)** and **O(n)** will never reach the growth curve of **$O(n^2)$,** so we can drop those terms all together. So we want to keep the most dominant term, ie: the worst case rule 1, and we are left with $O(n^2)$ which is our solution to the time complexity of the above equation.

What about **O(log n)**?

These are simple, if you have a problem where the number of elements in the problem space are halved you're going to have a **log n** in the runtime somewhere. A basic example is a binary search algorithm. After each search the problem search space is halved, thus decreasing the time it will take to get to the solution.

This should be a mile high view on runtimes to get you thru the simpler problems.