BIG O NOTATION

What is BIG O? simplified analysis of an algorithm’s efficiency

1. Complexity in terms of input size, N
2. Machine- independent
3. Basic computer steps

Types of measurement

1. Worst-case
2. Best-case
3. Average-case

General rules

1. Ignore constants.
   1. 5n -> O(n)
2. Certain terms “dominate” others.

O(1) < O(logn) < O(n) < O(nlogn) < O(n2) < O(2n) < O(n!)

i.e ignore lower terms

We say that an algorithm is **O(f(n))** if the number of simple operations the computer has to do is eventually less than a constant times **f(n)**, as **n** increases

* f(n) could be linear (f(n) = n)
* f(n) could be quadratic (f(n) = n2)
* f(n) could be constant (f(n) = 1)
* f(n) could be something entirely different!

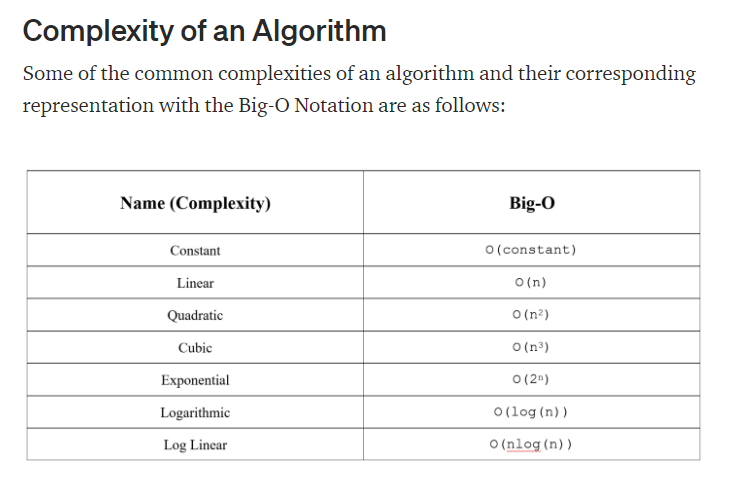
Rules of Thumb

1. Constants Don’t Matter
2. Smaller Terms Don’t Matter

Big O Shorthands

1. Analyzing complexity with big O can get complicated.
2. There are several rules of thumb that can help.
3. These rules won’t ALWAYS work, but are helpful starting point.

* Arithmetic Operations are constant.
* Variable assignment is constant.
* Accessing elements in an array (by index) or object (by key) is constant.
* In a loop, the complexity is the length of the loop times the complexity of whatever happens inside of the loop.



Let’s go through some commonly used bigO notations and their complexity and understand them a little better.

* + O(1) : Describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set

Function firstItem(arr) { return arr[0];}

The above function firstItem(), will always take the same time to execute, as it returns the first item from an array, irrespective of its size. The running time of this function is independent of input size, and so it has a constant complexity of O(1).

Relating it to the above explanation, even in the worst case scenario of this algorithm (assuming input to be extremely large), the running time would remain constant and not go beyond a certain value. So, its BigO complexity is constant, that is O(1).

* + O(N): Describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. Take a look at the example below. We have a function called *matchValue(*) which returns true whenever a matching case is found in the array. Here, since we have to iterate over the whole of the array, the running time is directly proportional to the size of the array.

function matchValue(arr, k){ for(var i = 0; i < arr.length; i++){ if(arr[i]==k){ return true; } else{ return false; } } }

This also demonstrates how Big O favors the worst-case performance scenario. A matching case could be found during any iteration of the for loop and the function would return early. But Big O notation will always assume the upper limit (worst-case) where the algorithm will perform the maximum number of iterations.

* + O(N²): This represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N³), O(N⁴), etc.

function containsDuplicates(arr){ for (var outer = 0; outer < arr.length; outer++){ for (var inner = 0; inner < arr.length; inner++){ if (outer == inner) continue; if (arr[outer] == arr[inner]) return true; } } return false;}

* + O(N²): This represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N³), O(N⁴), etc.

function recursiveFibonacci(number){ if (number <= 1) return number; return recursiveFibonacci(number - 2) + recursiveFibonacci(number - 1);}



Space Complexity in JS (Rules of Thumb)

* + Most primitives (Booleans, numbers, undefined, null) are constant space.
  + Strings require O(n) space (where n is the string length).
  + Reference types are generally O(n), where n is the length (for arrays) or the number of keys (for objects).

**Logarithm is inverse of Exponentiation.**

**Log2(8) = 3 ----🡪 238 What power give us 8 -> log2(value) = exponent -> 2exponent = value**