Big O

Good code is readable and scalable (think big O for scalability).

Scalability = Speed & Memory or Time & Space.

"3 Pillars of Programming"

1. Readable

2. Speed (Time Complexity)

3. Memory (Space Complexity)

\*\*There is generally a tradeoff between Time & Space when coding solutions to problems.

O(1) = Constant Time. Does not matter the size of input, always performs one operation. Another common, generally.

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

O(n) = Linear Time. Based on size of input and number of operations. Most common, generally.

O(n log(n)) Log Linear. Usually sorting operations, Divide & Conquer.

O(n^2) = Quadratic Time. Nested loops. Every element in a collection needs to be compared to every other element.

O(2^n) Exponential Time. Recursive algorithms that solves a problem of size N.

O(n!) = Factorial Time. Adding a loop for every element. Not common and should avoid.

What causes time complexity:

Operations (+, -, \*, /)

Comparisons (<, >, ==)

Looping (for, while)

Outside Function call (function())

What causes space complexity:

Variables

Data Structures

Function Call

Allocations

\*\*Heap is where variable assignments reside

\*\*Stack is where function calls reside

Big O Rules:

1. Always assume worst case.

2. Remove constants.

3. Different terms for input.

-If a function has two inputs and the function has two loops (non-nested) that loops through each input, this would be O(n+k).

-If the function had one input but 2 loops (non-nested), then it would just be O(n).

-If the function has nested loops & one input, then O(n\*n) or O(n^2).

-If the function has nested loops and two inputs within the nested loops, then O(a\*b).

-Any functions that have loops on the same indentation = adding.

-Any functions that have loops within a loop = multiplying.

4. Drop non-dominants. Keep most dominant term (most significant term).

-Ex. O(n + n^2) = O(n^2).

-Ex. O(x^2+3x+100+x/2) = O(x^2)