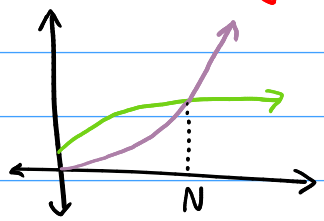
Algorithm Analysis

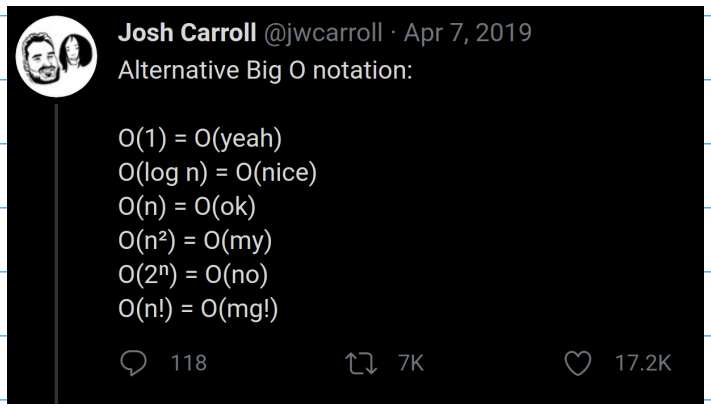
Big-O Notation

Given two functions f and g, we say that $f(n)$ is $O(g(n))$ if f is eventually bounded from above by g.



After a certain point, the purple function is always going to be greater than the green one.

Mathematical def'n: $f(n)$ is $O(g(n))$ if there is a constant C and some number N such that $f(n) \leq Cg(n)$ for all $n \geq N$.



Examples

- $2n^2+n+3$ is $O(n^2)$, but not $O(n)$

  Lesson: Lower order terms don't matter with big-O

- $2^n + n^{1000000}$ is $O(2^n)$, but not $O(n^{1000000})$

  Lesson: Exponential functions always grow faster than polynomials

- $\log n$ is $O(n)$ and $O(n^{1/2})$ and $O(n^{0.0000001})$, but not $O(1)$

  Lesson: Logarithms always grow slower than polynomials but faster than constants

Complexity

An algorithm's time complexity is a measure of how many primitive operations (array accesses, arithmetic, etc.) are required with respect to the size of its input.

One defines space complexity in a similar way for how much additional space/memory is used by the algorithm.

Usually we do worst-case analysis.