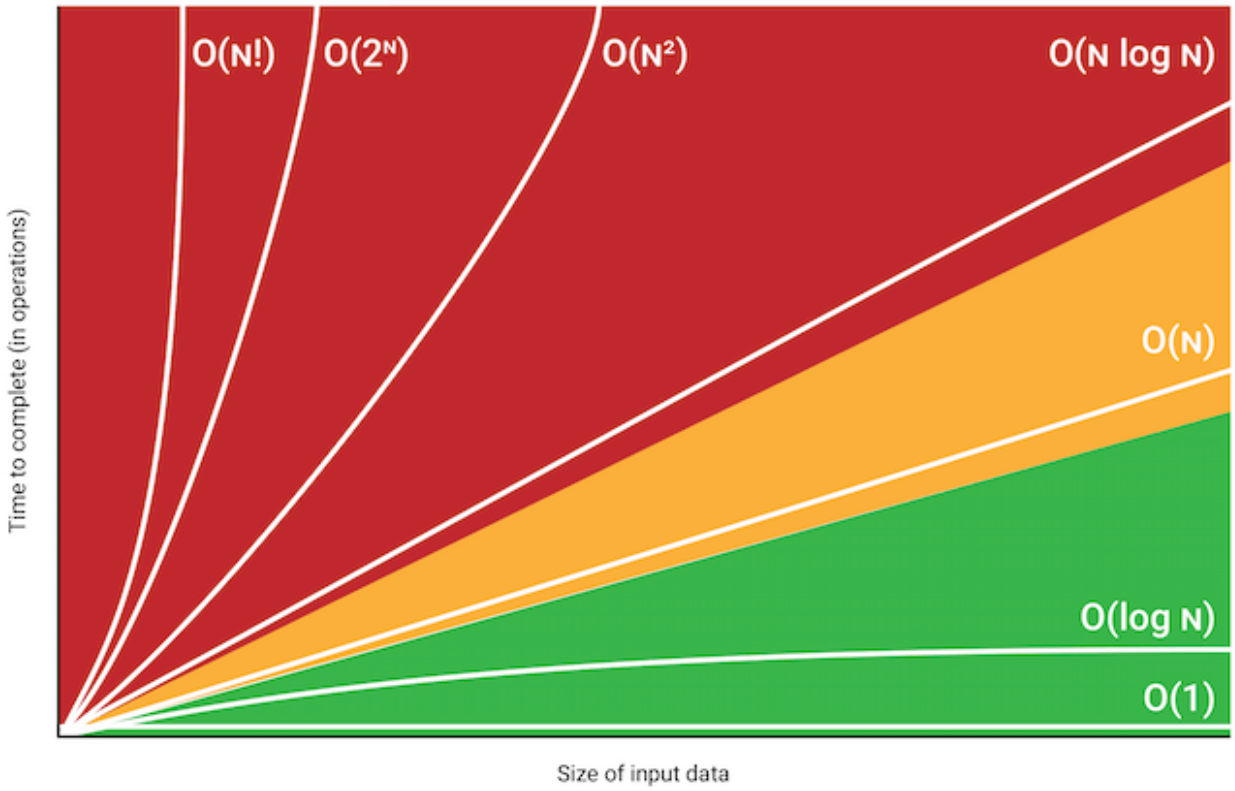
Big-O Notation Explained



The entire point of Big-O notation is to be able to compare how efficiently one algorithm solves big problems compared to another.

Big-O notation is a metric for algorithm scalability. So, if you have a giant list of names, or phone numbers, or some other kind of input—what is the best viable way of sorting them, or searching them, based on whether the list has 10 entries, or 100, or a million?

That is what Big-O notation tells you. It tells you how well that approach will do against large problems.

O (1) is pronounced as /' Oh of one '/ which is *constant complexity*.

Which means, no matter what you provide as input to the algorithm, it will still run in the same amount of time.

- 1 item, 1 *second*
- 10 items, 1 *second*
- 100 items, 1 *second*

O (log n) is pronounced as /'oh log en'/ which is *logarithmic complexity*.

Which means, the calculation time barely increases as you exponentially increase the input numbers.

- 1 item, 1 *second*
- 10 items, 2 *second*
- 100 items, 3 *second*

O (n^2) is pronounced as /'oh of en squared'/ which is *quadratic complexity*.

Which means, the calculation time increases at the pace of n^2.

- 1 item, 1 *second*
- 10 items, 100 second (about 1 and a half minutes)
- 100 items, 10,000 second (about 3 hours)

O (n!) /'oh, of en factorial'/ which is *factorial complexity*.

The calculation time increases at the pace of n! which means if n is 5, it is 5x4x3x2x1, or 120. This is not so bad at low values of n, but it quickly becomes impossible.

- N=1, 1 *option*
- N=10, 3,628,800 *options*
- N=100, 9.3326215400x10157 *options*

Summary

1. Algorithms are lists of steps for solving problems.
2. Some algorithms are good at problems when they're small, but fail at scale, e.g., with exceptionally long lists of things to sort or search.
3. Big-O notation is the way to tell how good a given algorithm is at solving exceptionally large problems.