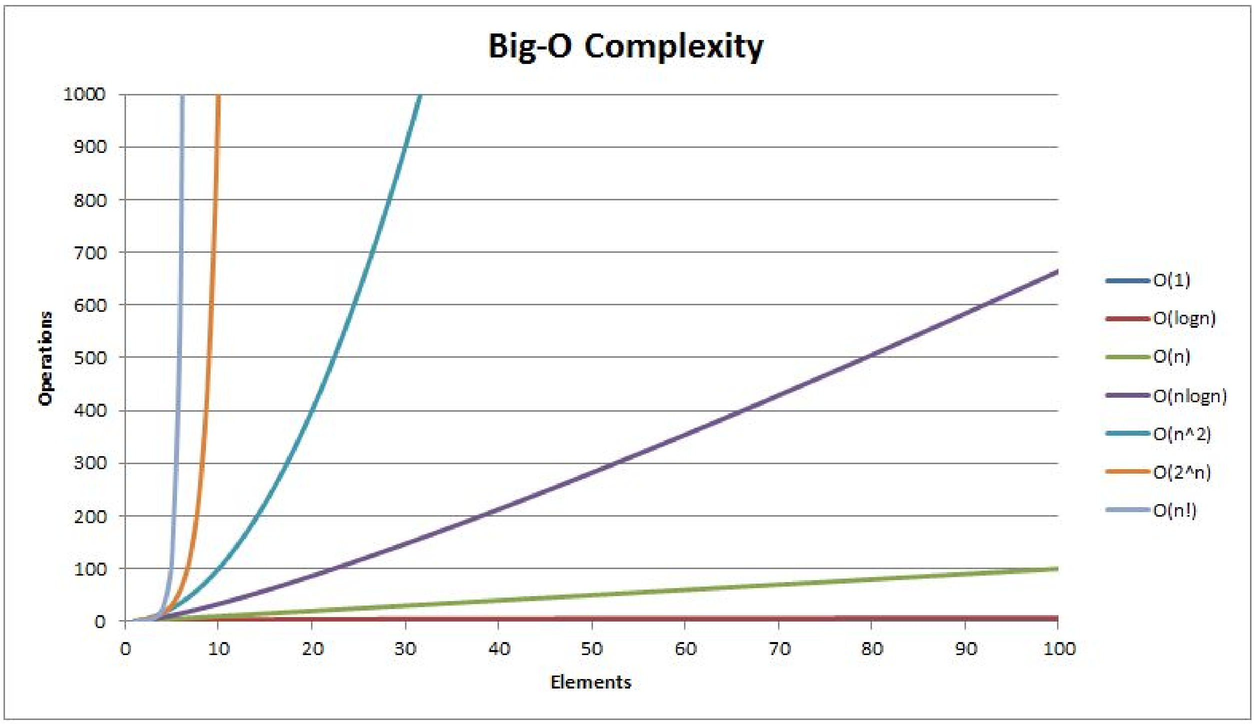
Big-Oh notes

Prof Bill, Jan 2020

“Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.”

- en.wikipedia.org/wiki/Big_O_notation

What function best describes the performance of your algorithm for N items?
/ my favorite chart of the course */*



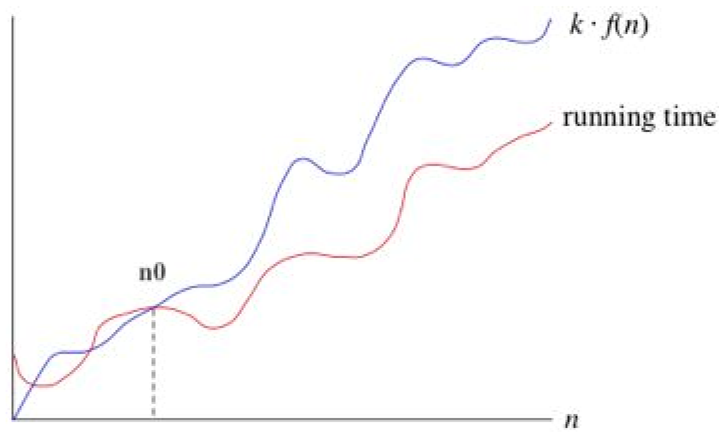
Source:www.hackerearth.com/practice/notes/big-o-cheatsheet-series-data-structures-and-algorithms-with-thier-complexities-1/

Seven performance categories are most common, for a problem of size = n:

- O(1) - constant time
- O(log(n)) - logarithmic time
- O(n) - linear time
- O(n log(n)) - quasi-linear or “n log n” time
- O(n^2) - polynomial time
- O(2^n) - exponential time
- O(n!) - factorial time

Formally, for O(f(n)) defines a function f(n) where:

```
running time <= k * f(n), for n > n0
```



For this Big-Oh, f(n) defines an **asymptotic limit** of our running time.

Constants, multipliers, and lower-order terms are ignored. Why? Because they are insignificant compared to the performance function for large N.

Example: Ignore constants, multipliers, and lower-order terms.

$$f(n) = 5n^2 + 7n + 101 \text{ is } O(n^2)$$

Try: Each Big-Oh function above for (piddly) N=100...function dominates growth.

Try: What is Big-Oh for the array operations: add, get, remove?

Big-Oh is **not** program timing or running benchmarks. It is a theoretical estimate, independent of specific program or computer.

Links:

- ➔ Another fun Big-OH summary, bigocheatsheet.com
- ➔ Wikipedia summary, en.wikipedia.org/wiki/Big_O_notation

Sedgewick Algorithms

Read: Section 4.1 of Sedgewick Java text, introcs.cs.princeton.edu/java.

Read: Section 1.4 Analysis of Algorithms of Sedgewick algorithms text, algs4.cs.princeton.edu.

Sedgewick's stopwatch example is **not** Big-Oh. That's benchmarking: write a program, create some test cases, run them, and time the results.

Note: Sedgewick uses an important Java API method at the heart of his stopwatch code, introcs.cs.princeton.edu/java/stdlib/Stopwatch.java.html.

```
System.currentTimeMillis();
```

docs.oracle.com/javase/8/docs/api/java/lang/System.html

Big-Oh is a theoretical estimation of your algorithm's performance. There are BIG advantages to this over benchmarking:

Algorithm analysis, Big-Oh/ Advantage	**Benchmarking**
Algorithm on paper + much less work/detail	Must write a complete program
Expected or worst case + much less work + important bounds on performance	Must develop a suite of test cases
Analysis independent of environment + much less work + theoretical bounds are key	Real world worries: CPU, operating system, language, network speed, etc

Important: How does Big-Oh justify all these shortcuts: see my fave chart on page 1. The difference in Big-Oh functions as N gets large is profound!

Note: Big-Oh can be applied to other resources as well: memory, disk space, etc.

Tilde approximations - throw away low-order terms that complicate formulas and aren't important as N gets large. Example:

$O(N^2 + 7N) =\sim O(N^2)$

Amortized analysis - spreading out the cost of an operation over a sequence of operations. The classic example, ArrayList...resizing is O(1) because it happens once for every N elements we add to the ArrayList:

> *In the resizing-array implementation of Bag, Stack, and Queue, starting from an empty data structure, any sequence of N operations takes time proportional to N in the worst case (amortized constant time per operation).*

/* Sedgewick spends some time counting bytes...again, that is more benchmarking, not algorithm analysis. This is important in improving a specific program, but isn't important to us in our study of Big-Oh. */

Sedgewick's cheatsheet is fantastic, algs4.cs.princeton.edu/cheatsheet.

I'll only ask you about Big-Oh, and it is most important...but there are other "Bigs".

Name	**Notation**	**Notes**
Big-Oh	$f(n)$ is $O(g(n))$	upper bound on performance
Tilde	$f(n) \sim g(n)$	equal to, asymptotically
Big-Omega	$f(n)$ is $\Omega(g(n))$	lower bound
Big-Theta	$f(n)$ is $\Theta(g(n))$	"tight", upper and lower bound

I really like this Big-Oh summary. Look at the code frags.

Common orders of growth.

NAME	NOTATION	EXAMPLE	CODE FRAGMENT
Constant	$O(1)$	array access arithmetic operation function call	`op();`
Logarithmic	$O(\log n)$	binary search in a sorted array insert in a binary heap search in a red–black tree	`for (int i = 1; i <= n; i = 2*i)` `   op();`
Linear	$O(n)$	sequential search grade-school addition BFPRT median finding	`for (int i = 0; i < n; i++)` `   op();`

Linearithmic	$O(n \log n)$	mergesort
		heapsort
		fast Fourier transform
			`for (int i = 1; i <= n; i++)`
			`   for (int j = i; j <= n; j = 2*j)`
			`      op();`

Quadratic	$O(n^2)$	enumerate all pairs
		insertion sort
		grade-school multiplication
			`for (int i = 0; i < n; i++)`
			`   for (int j = i+1; j < n; j++)`
			`      op();`

Cubic	$O(n^3)$	enumerate all triples
		Floyd–Warshall
		grade-school matrix multiplication
			`for (int i = 0; i < n; i++)`
			`   for (int j = i+1; j < n; j++)`
			`      for (int k = j+1; k < n; k++)`
			`         op();`

Polynomial	$O(n^c)$	ellipsoid algorithm for LP AKS primality algorithm Edmond's matching algorithm	
Exponential	$2^{O(n^c)}$	enumerating all subsets enumerating all permutations backtracing search	

Source: algs4.cs.princeton.edu/cheatsheet