Big O

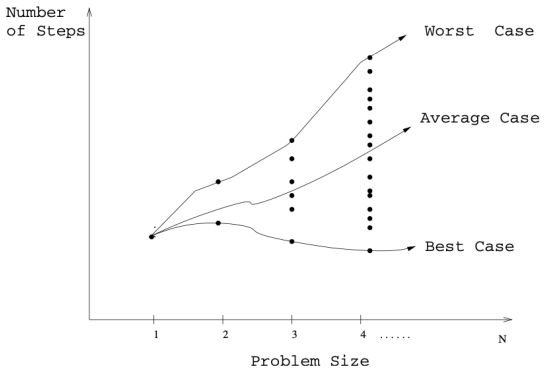
CS Osvita

January 14, 2025

Big O	Rank	Meaning
O(1)	😎	Speed doesn't depend on the size of the dataset
O(log n)	😄	10x the data means 2x more time
O(n)	😐	10x the data means 10x more time
O(n log n)	😖	10x the data means about 20x more time
O(n^2)	😫	10x the data take 100x more time
O(2^n)	😱	The dilithium crystals are breaking up!

Understanding this is key for technologists

To understand how good or bad an algorithm is, we must know how it works over all instances. For the sorting problem, the set of possible input instances consists of all possible arrangements of n keys over all possible values of n. We can represent each input instance as a point on a graph where the x-axis represents the size of the input problem (for sorting, the number of items to sort), and the y-axis denotes the number of steps taken by the algorithm in this instance.



We can define three interesting functions over the plot of these points:

1. The **worst-case** complexity of the algorithm is the function defined by the maximum number of steps taken in any instance of size n. This represents the curve passing through the highest point in each column.

2. The **best-case** complexity of the algorithm is the function defined by the minimum number of steps taken in any instance of size n. This represents the curve passing through the lowest point of each column.

3. The **average-case** complexity of the algorithm is defined by the average number of steps over all instances of size n.

The critical thing to realize is that each of these time complexities defines a numerical function, representing time versus problem size. These functions are as well defined as any other numerical function, be it y = $x^2 - 2x + 1$ or the price of Google stock as a function of time. However, time complexities are such complicated functions that we must simplify them to work with them. For this, we need the "Big Oh" notation.

Notation

It is challenging to work precisely with these functions because they tend to:

- Have too many bumps – an algorithm such as binary search typically runs a bit faster for arrays of size exactly $n = 2^k - 1$ because the array partitions work out nicely. This detail is not particularly significant, but it warns us that any algorithm's exact time complexity function can be very complicated, with little up and down bumps.

- Require too much detail to specify precisely – counting the exact number of RAM instructions executed in the worst case requires the algorithm to be specified to the detail of a complete computer program. Further, the precise answer depends upon uninteresting coding details (e.g., did he use a case statement or nested ifs?). Performing a precise worst-case analysis like:

  $$T(n) = 12754n^2 + 4353n + 834log n + 13546$$

  would be difficult work but provides us little extra information than the observation that "the time grows quadratically with n".

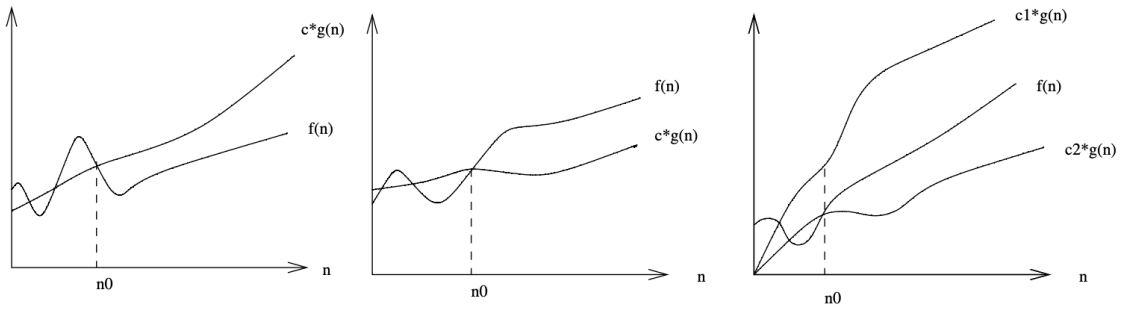
Using the Big Oh notation, it is much easier to talk in terms of simple **upper** and **lower** bounds of time-complexity functions. The Big Oh simplifies our analysis by ignoring levels of detail that do not impact our comparison of algorithms.

The Big Oh notation ignores the difference between multiplicative constants. The functions $f(n) = 2n$ and $g(n) = n$ are identical in Big Oh analysis. This makes sense, given our application. Suppose a given algorithm in C language

ran twice as fast as one with the same algorithm written in *Java*. This multiplicative factor of two tells us nothing about the algorithm since both programs implement the same algorithm. We ignore such constant factors when comparing two algorithms. The formal definitions associated with the Big Oh notation are as follows:

- $f(n) = O(g(n))$ means $c * g(n)$ is an upper bound on $f(n)$. Thus there exists some constant c such that $f(n) \leq cg(n)$, for large enough n (i.e. $n \geq n0$ for some constant n0).
- $f(n) = Omega(g(n))$ means $c * g(n)$ is a lower bound on f(n). Thus there exists some constant c such that $f(n) \geq c * g(n)$, for all $n \geq n0$.
- $f(n) = Theta(g(n))$ means $c_1 * g(n)$ is an upper bound on f(n) and $c_2 * g(n)$ is a lower bound on f(n), for all $n >= n0$. Thus there exist constants c_1 and c_2 such that $f(n) \leq c_1 * g(n)$ and $f(n) \geq c_2 * g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

Got it? These definitions are illustrated in the following image. Each definition assumes a constant $n0$ beyond which they are always satisfied. We are not concerned about small values of n (i.e., anything to the left of n_0). After all, we don't care whether one sorting algorithm sorts six items faster than another but seek which algorithm proves faster when sorting 10,000 or 1,000,000 items. The Big Oh notation lets us ignore details and focus on the big picture.



Examples:

$3n^2 - 100n + 6 = O(n^2)$, because I choose $c = 3$ and $3n^2 > 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 = O(n^3)$, because I choose $c = 1$ and $n^3 > 3n^2 - 100n + 6$ when $n > 3$;

$3n^2 - 100n + 6 \neq O(n)$, because for any c I choose $c \times n < 3n^2$ when $n > c$;

$3n^2 - 100n + 6 = \Omega(n^2)$, because I choose $c = 2$ and $2n^2 < 3n^2 - 100n + 6$ when $n > 100$;

$3n^2 - 100n + 6 \neq \Omega(n^3)$, because I choose $c = 3$ and $3n^2 - 100n + 6 < n^3$ when $n > 3$;

$3n^2 - 100n + 6 = \Omega(n)$, because for any c I choose $cn < 3n^2 - 100n + 6$ when $n > 100c$;

$3n^2 - 100n + 6 = \Theta(n^2)$, because both O and Ω apply;

$3n^2 - 100n + 6 \neq \Theta(n^3)$, because only O applies;

$3n^2 - 100n + 6 \neq \Theta(n)$, because only Ω applies.

The Big O notation provides a rough notion of equality when comparing functions. It is somewhat jarring to see an expression like $n^2 = O(n^3)$, but its meaning can always be resolved by returning to the definitions of upper and lower bounds. It is perhaps most instructive to read the "=" here as one of the functions. Clearly, n^2 is one of functions that are $O(n^3)$. The visual explanation of big-O can be found here.