Asymptotic Notation (Big Oh)

January 2014 BJ Keller

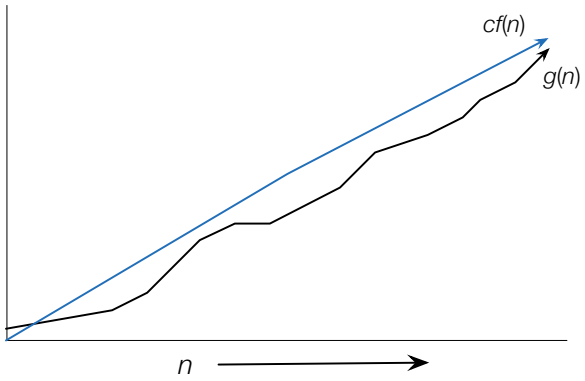
When we talk about algorithms in Computer Science we usually refer to their efficiency with expressions using asymptotic notation, and almost always use the Big Oh notation. The notation comes from mathematics, but our use of it is from the study of algorithms.

Algorithms: An *algorithm* defines a finite series of computational steps that solve a given problem, where the *problem* defines the input and the exact output. A software system may use a number of different algorithms, which solve much more specific problems than the whole system will. Classic examples are sorting and searching a collection of values. In most modern languages, these are already implemented in an API, and often we just need to decide which alternative we might use based on factors such as efficiency.

Efficiency: When we consider the efficiency of an algorithm, we will usually consider either how much time or space it requires as a function of the size of the input. For instance, if we are sorting a sequence of values stored in an array, the size of the input is the number of values, which we usually write as n. We could, but don't usually do experiments to determine how long a sort on an array of length n takes. Instead, we do analysis of the operations involved to derive a function on n, and then use the asymptotic notation to focus on the major components of the time or space cost. For instance, with sorting an array, we would count how many times we swap values for a given input array. The notation allows us to ignore the precise time required for each swap, as well as ignoring the lower cost steps of the algorithm.

Cases: Efficiency is stated relative to a particular scenario for the input, most commonly the worst case. The worst case is a scenario in which the algorithm's performance in terms of time or space is maximized. In searching a list for a value, it would be that the value is the very last element. Sometimes we'll want to refer to the best and average cases, which requires that we delve into algorithm analysis.

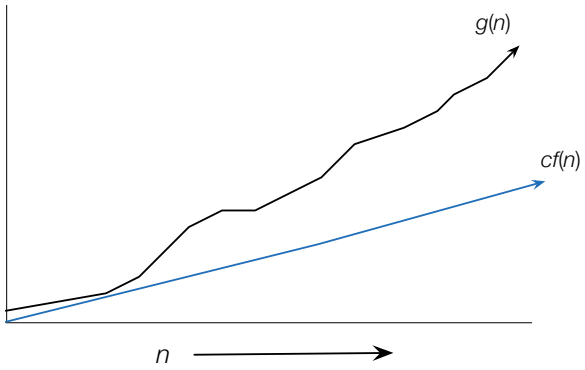
Big Oh: In defining the notation, $f(n)$ and $g(n)$ refer to (positive) integer valued functions on n. The Big Oh notation $O(f(n))$ refers to a set of functions where $g(n)$ is in $O(f(n))$ if and only if there exists a positive constant c such that $g(n) \leq cf(n)$ for $n \geq n_0$, for some $n_0 \geq 0$. The set $O(f(n))$ consists of all of the functions on n that can be bounded above by $f(n)$. Members of $O(n^2)$ include $3n^2$, $3n^2+4n+5$, and $2117n^2$. (The first and last functions are their own bounds, and the middle function is less than $5n^2$.)



Function g(n) in O(f(n)) means that f(n) is an upper bound of g(n) when multiplied by a constant.

You might see the notation used in a statement like "the implementation of sort(-) is guaranteed to take time on the order of $O(n \log_2 n)$ in the worst case", which simply means that the time function for the worst possible input scenario is in the set $O(n \log_2 n)$.

Big Oh just says that there is an upper bound and doesn't guarantee that it is tight – the function $5913n$ is in both $O(n)$ and $O(n^2)$. If we want to be more precise, we also need to use lower bounds, which is the Big Omega notation. The notation $\Omega(f(n))$ refers to a set of functions where $g(n)$ is in $\Omega(f(n))$ whenever there exists a positive constant c such that $g(n) \geq cf(n)$ for $n \geq n_0$, for some $n_0 \geq 0$. So, the Big Omega notation defines the set of functions that have a specific function as a *lower* bound.



Function g(n) in Ω(f(n)) means that f(n) is an lower bound of g(n) when multiplied by a constant.

Big Omega has the same problem for being precise as Big Oh: $391n^2$ is in both $\Omega(n)$ and $\Omega(n^2)$. However, if we require both, we can narrow things down, and so we

say $g(n)$ is in $\Theta(f(n))$ when $g(n)$ is in both $\Omega(f(n))$ and $O(f(n))$. Of course, nearly all of the standard algorithms you will see in an API have time bounds of this kind, so, in this circumstance, you can almost always read statements involving Big Oh as meaning Big Theta.

Functions: Common functions you'll see in reference to algorithms are O(1), $O(\log_2 n)$, $O(n)$, $O(n \log_2 n)$, $O(n^2)$. Instead of referring to these functions this way, we might say an algorithm takes constant time, logarithmic time, linear time, log linear time, or quadratic time.

Algorithm Analysis: When we analyze algorithms we usually do some form of counting (as in combinatorics) to come up with a function of n to describe time or space efficiency. Frequently, this is simple enough, but depending on the algorithm may involve relatively involved math.

An example that is simple would be linear search for a value v in a sequence of n values stored in an array a:

```
i = 0
for i ≤ n and a[i] != v
  i = i + 1
```

The array has n values, and in the worst case the value v is either the last element, or simply not in the array. Let's start by assuming v is the last entry and count everything. First, we have the initialization of i, then all but the last iteration of the loop has 2 comparisons, and one assignment. The last iteration will perform both comparisons and skip the assignment. So, we have 1+ (n-1) = n assignments, and $2n$ comparisons, for a total of $3n$ operations. This is in $\Theta(n)$.

If we considered the case that v is not in the array, then the numbers are slightly different: $n+1$ assignments, and $2n+2$ comparisons, or $3n+3$. But, we can show that is also in $\Theta(n)$. The message is that we need not work so hard counting!

In particular, we can generally choose one type of operation to count, and not worry about the others. In this case, we could have counted the number of times the condition was checked and it would give us our function. The only time we might need to worry is if there is more than one input where the algorithm is dependent on the size.

Knowing the worst case can be helpful in deciding between algorithms, but we might consider an algorithm that has a bad worst case time but a reasonable average case time, especially if the worst case is sufficiently unlikely. Let's try it for linear search: since the value could be anywhere, the average is the sum of all possible positions divided by the number of positions. The sum of j, for j = 1 to n, is a standard arithmetic sum with closed form $n(n+1)/2$. The average is $(n+1)/2$, which is in $\Theta(n)$. In this case, it didn't work out, but some algorithms for famously complex problems have much better average case time functions than the worst case.

Since most of the standard algorithms that we use in programming are for a small part of our program and are often repeated, we want them to be more efficient. So, linear time for a search is actually not good enough. If the array is sorted, we can guarantee a search time of $O(\log_2 n)$ using binary search:

```
left = 0, right = n
mid = (left+right)/2
while right != left+1 and a[mid] != v
   if v < a[mid]
      right = mid
   else
      left = mid
   mid = (left+right)/2
```

The algorithm uses the strategy of divide and conquer. Specifically, since we know the array is sorted, as soon as we know whether the search value is less than or greater than the middle value we can eliminate the other half. In the worst case, the value v is the very last place we look, which corresponds to the number of times we can divide regions of the array in half. This is a positive number d such that $2^d = n$, which is precisely $\log_2 n$. Not a proof, but sufficient hand waiving to understand why binary search is $\Theta(\log_2 n)$. Of course, the array has to be sorted, which is in $\Theta(n \log_2 n)$. We can do better with hashing, but that is another story.

Resources:

1) bigocheatsheet
2) S. Skiena, Algorithm Design Manual, Springer-Verlag, 2008. (site: algorist.com)

Exercises:

1) You have a problem in your program that is solved by two functions in the API called quibble_this_way(-) and quibble_that_way(-). Both take worst case time in $O(f(n))$ for some function $f(n)$. What does this mean in terms of choosing the most efficient algorithm for your program?
2) There is a sorting algorithm that takes time in $O(n^2)$, while the other algorithms take time in $O(n \log_2 n)$. How do you choose the most efficient algorithm?
3) You have two ways to store a set of values. The worst case search time of the first is $O(n)$ with space overhead of n, and the search time of the other is $O(\log_2 n)$ with space overhead of $2n$. (*Overhead* is the storage needed beyond the actual data.) In what circumstances might you select the first over the second, and the second over the first?