# Asymptotic Notation

## Introduction

A problem may have numerous algorithmic solutions. In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run. Or, more accurately, you need to be able to judge how long two solutions will take to run, and choose the better of the two. You don't need to know how many minutes and seconds they will take, but you do need some way to compare algorithms against one another.

*Asymptotic complexity* is a way of expressing the *main component* of the cost of an algorithm, using idealized units of computational work. Consider, for example, the algorithm for sorting a deck of cards, which proceeds by repeatedly searching through the deck for the lowest card. The asymptotic complexity of this algorithm is the *square* of the number of cards in the deck. This quadratic behavior is the main term in the complexity formula, it says, e.g., if you double the size of the deck, then the work is roughly quadrupled.

The exact formula for the cost is more complex, and contains more details than are needed to understand the essential complexity of the algorithm. With our deck of cards, in the worst case, the deck would start out reverse-sorted, so our scans would have to go all the way to the end. The first scan would involve scanning 52 cards, the next would take 51, etc. So the cost formula is 52 + 51 + ... + 2. generally, letting N be the number of cards, the formula is 2 + ... + N, which equals $((N) * (N + 1) / 2) - 1 = ((N^2 + N) / 2) - 1 = (1 / 2)N^2 + (1 / 2)N - 1$. But the N^2 term dominates the expression, and this is what is key for comparing algorithm costs. (This is in fact an *expensive* algorithm; the best sorting algorithms run in sub-quadratic time.)

Asymptotically speaking, in the limit as N tends towards infinity, 2 + 3 + ... + N gets closer and closer to the pure quadratic function (1/2)N^2. And what difference does the constant factor of 1/2 make, at this level of abstraction? So the behavior is said to be O($n^2$).

Now let us consider how we would go about *comparing* the complexity of two algorithms. Let f(n) be the cost, in the worst case, of one algorithm, expressed as a function of the input size n, and g(n) be the cost function for the other algorithm. E.g., for sorting algorithms, f(10) and g(10) would be the maximum number of steps that the algorithms would take on a list of 10 items. If, for all values of n >= 0, f(n) is less than or equal to g(n), then the algorithm with complexity function f is strictly faster. But, generally speaking, our concern for computational cost is for the cases with large inputs; so the comparison of f(n) and g(n) for small values of n is less significant than the "long term" comparison of f(n) and g(n), for n larger than some threshold.

Note that we have been speaking about *bounds* on the performance of algorithms, rather than giving exact speeds. The actual number of steps required to sort our deck of cards (with our naive quadratic algorithm) will depend upon the order in which the cards begin. The actual time to perform each of our steps will depend upon our processor speed, the condition of our processor cache, etc., etc. It's all very complicated in the concrete details, and moreover not relevant to the essence of the algorithm.

# Big-O Notation

## Definition

Big-O is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.

More formally, for non-negative functions, *f(n)* and *g(n)*, if there exists an integer $n_0$ and a constant $c > 0$ such that for all integers $n > n_0$, $f(n) \leq cg(n)$, then *f(n)* is Big O of *g(n)*. This is denoted as "$f(n) = O(g(n))$". If graphed, *g(n)* serves as an upper bound to the curve you are analyzing, *f(n)*.

Note that if *f* can take on finite values only (as it should happen normally) then this definition implies that there exists some constant *C* (potentially larger than *c*) such that for all values of *n*, $f(n) \leq Cg(n)$. An appropriate value for *C* is the maximum of *c* and .

## Theory Examples

So, let's take an example of Big-O. Say that f(n) = 2n + 8, and g(n) = $n^2$. Can we find a constant $n_0$, so that 2n + 8 <= $n^2$? The number 4 works here, giving us 16 <= 16. For any number n greater than 4, this will still work. Since we're trying to generalize this for large values of n, and small values (1, 2, 3) aren't that important, we can say that f(n) is generally faster than g(n); that is, f(n) is bound by g(n), and will always be less than it.

It could then be said that f(n) runs in O($n^2$) time: "f-of-n runs in Big-O of n-squared time".

To find the upper bound - the Big-O time - assuming we know that f(n) is equal to (exactly) 2n + 8, we can take a few shortcuts. For example, we can remove all constants from the runtime; eventually, at some value of c, they become irrelevant. This makes f(n) = 2n. Also, for convenience of comparison, we remove constant multipliers; in this case, the 2. This makes f(n) = n. It could also be said that f(n) runs in O(n) time; that lets us put a tighter (closer) upper bound onto the estimate.

## Practical Examples

O(n): printing a list of n items to the screen, looking at each item once.

O(ln n): also "log n", taking a list of items, cutting it in half repeatedly until there's only one item left.

O($n^2$): taking a list of n items, and comparing every item to every other item.

## *Big-Omega Notation*

For non-negative functions, *f(n)* and *g(n)*, if there exists an integer $n_0$ and a constant $c > 0$ such that for all integers $n > n_0$, $f(n) \geq cg(n)$, then *f(n)* is omega of *g(n)*. This is denoted as "$f(n) = \Omega(g(n))$".

This is almost the same definition as Big Oh, except that "$f(n) \geq cg(n)$", this makes *g(n)* a lower bound function, instead of an upper bound function. It describes the **best that can happen** for a given data size.

## Theta Notation

**Theta Notation**

For non-negative functions, *f(n)* and *g(n)*, *f(n)* is theta of *g(n)* if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. This is denoted as "$f(n) = \Theta(g(n))$".

This is basically saying that the function, *f(n)* is bounded both from the top and bottom by the same function, *g(n)*.

### Little-O Notation

For non-negative functions, *f(n)* and *g(n)*, *f(n)* is little o of *g(n)* if and only if $f(n) = O(g(n))$, but $f(n) \neq \Theta(g(n))$. This is denoted as "$f(n) = o(g(n))$".

This represents a loose bounding version of Big O. *g(n)* bounds from the top, but it does not bound the bottom.

### Little Omega Notation

For non-negative functions, *f(n)* and *g(n)*, *f(n)* is little omega of *g(n)* if and only if $f(n) = \Omega(g(n))$, but $f(n) \neq \Theta(g(n))$. This is denoted as "$f(n) = \omega(g(n))$".

Much like Little Oh, this is the equivalent for Big Omega. *g(n)* is a loose lower boundary of the function *f(n)*; it bounds from the bottom, but not from the top.

## How asymptotic notation relates to analyzing complexity

Temporal comparison is not the only issue in algorithms. There are space issues as well. Generally, a tradeoff between time and space is noticed in algorithms. Asymptotic notation empowers you to make that trade off. If you think of the amount of time and space your algorithm uses as a function of your data over time or space (time and space are usually analyzed separately), you can analyze how the time and space is handled when you introduce more data to your program.

This is important in data structures because you want a structure that behaves efficiently as you increase the amount of data it handles. Keep in mind though that algorithms that are efficient with large amounts of data are not always simple and efficient for small amounts of data. So if you know you are working with only a small amount of data and you have concerns for speed and code space, a trade off can be made for a function that does not behave well for large amounts of data.

## A few examples of asymptotic notation

Generally, we use asymptotic notation as a convenient way to examine what can happen in a function in the worst case or in the best case. For example, if you want to write a function that searches through an array of numbers and returns the smallest one:

```
function find-min(array a[1..n])

  let j :=
  for i := 1 to n:
    j := min(j, a[i])
  repeat
  return j
end
```

Regardless of how big or small the array is, every time we run **find-min**, we have to initialize the *i* and *j* integer variables and return *j* at the end. Therefore, we can just think of those parts of the function as constant and ignore them.

So, how can we use asymptotic notation to discuss the **find-min** function? If we search through an array with 87 elements, then the *for* loop iterates 87 times, even if the very first element we hit turns out to be the minimum. Likewise, for $n$ elements, the for loop iterates $n$ times. Therefore we say the function runs in time $O(n)$.

What about this function:

```
function find-min-plus-max(array a[1..n])
  // First, find the smallest element in the array
  let j := ∞;
  for i := 1 to n:
    j := min(j, a[i])
  repeat
  let minim := j

  // Now, find the biggest element, add it to the smallest and
  j := −∞;
  for i := 1 to n:
    j := max(j, a[i])
  repeat
  let maxim := j

  // return the sum of the two
  return minim + maxim;
```

```
end
```

What's the running time for **find-min-plus-max**? There are two *for* loops, that each iterate $n$ times, so the running time is clearly $O(2n)$. Because 2 is a constant, we throw it away and write the running time as $O(n)$. Why can you do this? If you recall the definition of Big-O notation, the function whose bound you're testing can be multiplied by some constant. If *f(x)* = *2x*, we can see that if *g(x)* = *x*, then the Big-O condition holds. Thus $O(2n) = O(n)$. This rule is general for the various asymptotic notations.