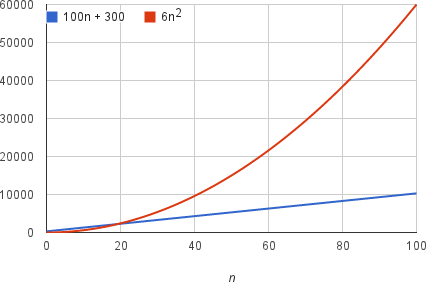
## Asymptotic notation

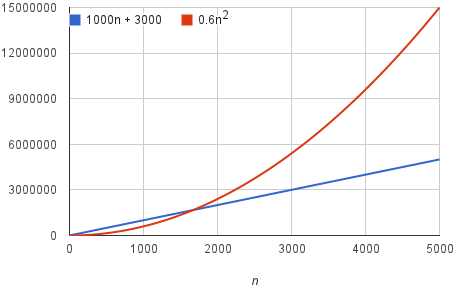
The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm. And that depends on the speed of the computer, the programming language, and the compiler that translates the program from the programming language into code that runs directly on the computer, as well as other factors.

Let's think about the running time of an algorithm more carefully. We use a combination of two ideas. First, we determine how long the algorithm takes, in terms of the size of its input. This idea makes intuitive sense, doesn't it? We've already seen that the maximum number of guesses in linear search and binary search increases as the length of the array increases. Or think about a GPS. If it knew about only the interstate highway system, and not about every little road, it should be able to find routes more quickly, right? So we think about the running time of the algorithm as a function of the size of its input.

The second idea is that we focus on how fast this function grows with the input size. We call that the **rate of growth** of the running time. To keep things manageable, we simplify the function to distill the most important part and cast aside the less important parts. For example, suppose that an algorithm, running on an input of size *n*, takes 6*n*​2​​+100*n*+300 machine instructions. The 6*n*​2​​ term becomes larger than the remaining terms, 100*n*+300, once *n* becomes large enough (20 in this case). Here's a chart showing values of 6*n*​2​​ and 100*n*+300 for values of *n* from 0 to 100:



We would say that the running time of this algorithm grows as*n*​2​​, dropping the coefficient 6 and the remaining terms 100*n*+300. It doesn't really matter what coefficients we use; as long as the running time is *an*​2​​+*bn*+*c*, for some numbers *a*>0, *b*, and *c*, there will always be a value of *n* for which *an*​2​​ is greater than *bn*+*c*, and this difference increases as *n* increases. For example, here's a chart showing values of 0.6*n*​2​​ and 1000*n*+3000, so that we've reduced the coefficient of *n*​2​​ by a factor of 10 and increased the other two constants by a factor of 10:



The value of *n* at which 0.6*n*​2​​ becomes greater than 1000*n*+3000 has increased, but there will always be such a crossover point, no matter what the constants.

By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth—without getting mired in details that complicate our understanding. When we drop the constant coefficients and the less significant terms, we use **asymptotic notation**. We'll see three forms of it: big-Θnotation, big-O notation, and big-Ω notation.

## Big-θ (Big-Theta) notation

Let's look at a simple implementation of linear search:

var doLinearSearch = function(array) {

for (var guess = 0; guess < array.length; guess++) {

if (array[guess] === targetValue) {

return guess; // found it!

}

}

return -1; // didn't find it

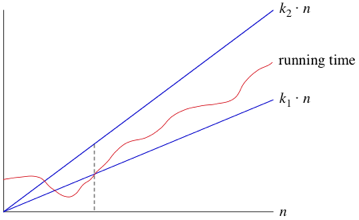
};

Let's denote the size of the array (array.length) by *n*. The maximum number of times that the for-loop can run is *n*, and this worst case occurs when the value being searched for is not present in the array. Each time the for-loop iterates, it has to do several things: compare guess with array.length; compare array[guess] with targetValue; possibly return the value of guess; and increment guess. Each of these little computations takes a constant amount of time each time it executes. If the for-loop iterates *n* times, then the time for all *n* iterations is *c*​1​​⋅*n*, where *c*​1​​ is the sum of the times for the computations in one loop iteration.

Now, we cannot say here what the value of *c*​1​​ is, because it depends on the speed of the computer, the programming language used, the compiler or interpreter that translates the source program into runnable code, and other factors. This code has a little bit of extra overhead, for setting up the for-loop (including initializing guess to 0) and possibly returning -1 at the end. Let's call the time for this overhead *c*​2​​, which is also a constant. Therefore, the total time for linear search in the worst case is *c*​1​​⋅*n*+*c*​

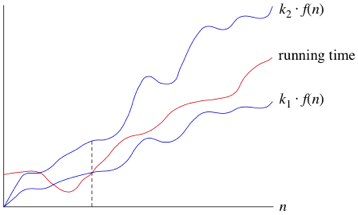
As we've argued, the constant factor *c*​1​​ and the low-order term *c*​2​​ don't tell us about the rate of growth of the running time. What's significant is that the worst-case running time of linear search grows like the array size *n*. The notation we use for this running time is Θ(*n*). That's the Greek letter "theta," and we say "big-Theta of *n*" or just "Theta of *n*."

When we say that a particular running time is Θ(*n*), we're saying that once *n* gets large enough, the running time is at least *k*​1​​⋅*n* and at most *k*​2​​⋅*n* for some constants *k*​1​​ and *k*​2​​. Here's how to think of Θ(*n*):



For small values of *n*, we don't care how the running time compares with *k*​1​​⋅*n* or *k*​2​​⋅*n*. But once *n* gets large enough—on or to the right of the dashed line—the running time must be sandwiched between *k*​1​​⋅*n* and *k*​2​​⋅*n*. As long as these constants *k*​1​​ and *k*​2​​ exist, we say that the running time is Θ(*n*).

We are not restricted to just *n* in big-Θ notation. We can use any function, such as *n*​2​​, *n*lg*n*, or any other function of *n*. Here's how to think of a running time that is Θ(*f*(*n*)) for some function *f*(*n*):



Once *n* gets large enough, the running time is between *k*​1​​⋅*f*(*n*) and *k*​2​​⋅*f*(*n*).

In practice, we just drop constant factors and low-order terms. Another advantage of using big-Θ notation is that we don't have to worry about which time units we're using. For example, suppose that you calculate that a running time is 6*n*​2​​+100*n*+300microseconds. Or maybe it's milliseconds. When you use big-Θ notation, you don't say. You also drop the factor 6 and the low-order terms 100*n*+300, and you just say that the running time is Θ(*n*​2​​).

When we use big-Θ notation, we're saying that we have an **asymptotically tight bound** on the running time. "Asymptotically" because it matters for only large values of *n*. "Tight bound" because we've nailed the running time to within a constant factor above and below.

## Functions in asymptotic notation

When we use asymptotic notation to express the rate of growth of an algorithm's running time in terms of the input size *n*, it's good to bear a few things in mind.

Let's start with something easy. Suppose that an algorithm took a constant amount of time, regardless of the input size. For example, if you were given an array that is already sorted into increasing order and you had to find the minimum element, it would take constant time, since the minimum element must be at index 0. Since we like to use a function of *n* in asymptotic notation, you could say that this algorithm runs in Θ(*n*​0​​) time. Why? Because *n*​0​​=1, and the algorithm's running time is within some constant factor of 1. In practice, we don't write Θ(*n*​0​​), however; we write Θ(1).

Now suppose an algorithm took Θ(log​10​​*n*) time. You could also say that it took Θ(lg*n*)time (that is, Θ(log​2​​*n*) time). Whenever the base of the logarithm is a constant, it doesn't matter what base we use in asymptotic notation. Why not? Because there's a mathematical formula that says



for all positive numbers *a*, *b*, and *n*. Therefore, if *a* and *b* are constants, then log​*a*​​*n* and log​*b*​​*n* differ only by a factor of log​*b*​​*a*, and that's a constant factor which we can ignore in asymptotic notation.

Therefore, we can say that the worst-case running time of binary search is Θ(log​*a*​​*n*) for any positive constant *a*. Why? The number of guesses is at most lg*n*+1, generating and testing each guess takes constant time, and setting up and returning take constant time. As a matter of practice, we write that binary search takes Θ(lg*n*) time because computer scientists like to think in powers of 2 (and there are fewer characters to write than if we wrote Θ(log​2​​*n*).)

There is an order to the functions that we often see when we analyze algorithms using asymptotic notation. If *a* and *b* are constants and *a*<*b*, then a running time of Θ(*n*​*a*​​)grows more slowly than a running time of Θ(*n*​*b*​​). For example, a running time of Θ(*n*), which is Θ(*n*​1​​), grows more slowly than a running time of Θ(*n*​2​​). The exponents don't have to be integers, either. For example, a running time of Θ(*n*​2​​) grows more slowly than a running time of Θ(*n*​2​​√​*n*​​​), which is Θ(*n*​2.5​​).

Logarithms grow more slowly than polynomials. That is, Θ(lg*n*) grows more slowly than Θ(*n*​*a*​​) for *any* positive constant *a*. But since the value of lg*n* increases as *n* increases, Θ(lg*n*) grows faster than Θ(1).

Here's a list of functions in asymptotic notation that we often encounter when analyzing algorithms, listed from slowest to fastest growing. This list is not exhaustive; there are many algorithms whose running times do not appear here:

1. Θ(1)
2. Θ(lg*n*)
3. Θ(*n*)
4. Θ(*n*lg*n*)
5. Θ(*n*​2​​)
6. Θ(*n*​2​​lg*n*)
7. Θ(*n*​3​​)
8. Θ(2​*n*​​)

Note that an exponential function *a*​*n*​​, where *a*>1, grows faster than any polynomial function *n*​*b*​​, where *b* is any constant.

Quizes

