**Algorithm Complexity Analysis**

<http://discrete.gr/complexity/>

**Introduction**

A lot of programmers that make some of the coolest and most useful software today, such as many of the stuff we see on the Internet or use daily, don't have a theoretical computer science background. They're still pretty awesome and creative programmers and we thank them for what they build.

However, theoretical computer science has its uses and applications and can turn out to be quite practical. In this article, targeted at programmers who know their art but who don't have any theoretical computer science background, I will present one of the most pragmatic tools of computer science: Big O notation and algorithm complexity analysis. As someone who has worked both in a computer science academic setting and in building production-level software in the industry, this is the tool I have found to be one of the truly useful ones in practice, so I hope after reading this article you can apply it in your own code to make it better. After reading this post, you should be able to understand all the common terms computer scientists use such as "big O", "asymptotic behavior" and "worst-case analysis".

This text is also targeted at the junior high school and high school students from Greece or anywhere else internationally competing in the [International Olympiad in Informatics](http://en.wikipedia.org/wiki/International_Olympiad_in_Informatics), an algorithms competition for students, or other similar competitions. As such, it does not have any mathematical prerequisites and will give you the background you need in order to continue studying algorithms with a firmer understanding of the theory behind them. As someone who used to compete in these student competitions, I highly advise you to read through this whole introductory material and try to fully understand it, because it will be necessary as you study algorithms and learn more advanced techniques.

I believe this text will be helpful for industry programmers who don't have too much experience with theoretical computer science (it is a fact that some of the most inspiring software engineers never went to college). But because it's also for students, it may at times sound a little bit like a textbook. In addition, some of the topics in this text may seem too obvious to you; for example, you may have seen them during your high school years. If you feel you understand them, you can skip them. Other sections go into a bit more depth and become slightly theoretical, as the students competing in this competition need to know more about theoretical algorithms than the average practitioner. But these things are still good to know and not tremendously hard to follow, so it's likely well worth your time. As the original text was targeted at high school students, no mathematical background is required, so anyone with some programming experience (i.e. if you know what recursion is) will be able to follow through without any problem.

Throughout this article, you will find various pointers that link you to interesting material often outside the scope of the topic under discussion. If you're an industry programmer, it's likely that you're familiar with most of these concepts. If you're a junior student participating in competitions, following those links will give you clues about other areas of computer science or software engineering that you may not have yet explored which you can look at to broaden your interests.

Big O notation and algorithm complexity analysis is something a lot of industry programmers and junior students alike find hard to understand, fear, or avoid altogether as useless. But it's not as hard or as theoretical as it may seem at first. Algorithm complexity is just a way to formally measure how fast a program or algorithm runs, so it really is quite pragmatic. Let's start by motivating the topic a little bit.

**Motivation**

We already know there are tools to measure how fast a program runs. There are programs called profilers which measure running time in milliseconds and can help us optimize our code by spotting bottlenecks. While this is a useful tool, it isn't really relevant to algorithm complexity. Algorithm complexity is something designed to compare two algorithms at the idea level — ignoring low-level details such as the implementation programming language, the hardware the algorithm runs on, or the instruction set of the given CPU. We want to compare algorithms in terms of just what they are: Ideas of how something is computed. Counting milliseconds won't help us in that. It's quite possible that a bad algorithm written in a low-level programming language such as [assembly](http://en.wikipedia.org/wiki/Assembly_language) runs much quicker than a good algorithm written in a high-level programming language such as[Python](http://www.python.org/) or [Ruby](http://www.ruby-lang.org/en/). So it's time to define what a "better algorithm" really is.

As algorithms are programs that perform just a computation, and not other things computers often do such as networking tasks or user input and output, complexity analysis allows us to measure how fast a program is when it performs computations. Examples of operations that are purely computational include numerical[floating-point operations](http://en.wikipedia.org/wiki/Floating_point) such as addition and multiplication; searching within a database that fits in RAM for a given value; determining the path an artificial-intelligence character will walk through in a video game so that they only have to walk a short distance within their virtual world (see **Figure 1**); or running a [regular expression](http://www.regular-expressions.info/)pattern match on a string. Clearly, computation is ubiquitous in computer programs.

Complexity analysis is also a tool that allows us to explain how an algorithm behaves as the input grows larger. If we feed it a different input, how will the algorithm behave? If our algorithm takes 1 second to run for an input of size 1000, how will it behave if I double the input size? Will it run just as fast, half as fast, or four times slower? In practical programming, this is important as it allows us to predict how our algorithm will behave when the input data becomes larger. For example, if we've made an algorithm for a web application that works well with 1000 users and measure its running time, using algorithm complexity analysis we can have a pretty good idea of what will happen once we get 2000 users instead. For algorithmic competitions, complexity analysis gives us insight about how long our code will run for the largest testcases that are used to test our program's correctness. So if we've measured our program's behavior for a small input, we can get a good idea of how it will behave for larger inputs. Let's start by a simple example: Finding the maximum element in an array.

**Counting instructions**

In this article, I'll use various programming languages for the examples. However, don't despair if you don't know a particular programming language. Since you know programming, you should be able to read the examples without any problem even if you aren't familiar with the programming language of choice, as they will be simple and I won't use any esoteric language features. If you're a student competing in algorithms competitions, you most likely work with [C++](http://www.cplusplus.com/doc/tutorial/), so you should have no problem following through. In that case I recommend working on the exercises using C++ for practice.

The maximum element in an array can be looked up using a simple piece of code such as this piece of [Javascript](http://www.quirksmode.org/js/intro.html) code. Given an input array A of size n:

var M = A[ 0 ];

for ( var i = 0; i < n; ++i ) {

if ( A[ i ] >= M ) {

M = A[ i ];

}

}

Now, the first thing we'll do is count how many fundamental instructions this piece of code executes. We will only do this once and it won't be necessary as we develop our theory, so bear with me for a few moments as we do this. As we analyze this piece of code, we want to break it up into simple instructions; things that can be executed by the CPU directly - or close to that. We'll assume our processor can execute the following operations as one instruction each:

* Assigning a value to a variable
* Looking up the value of a particular element in an array
* Comparing two values
* Incrementing a value
* Basic arithmetic operations such as addition and multiplication

We'll assume branching (the choice between if and else parts of code after the if condition has been evaluated) occurs instantly and won't count these instructions. In the above code, the first line of code is:

var M = A[ 0 ];

This requires 2 instructions: One for looking up A[ 0 ] and one for assigning the value to M (we're assuming that n is always at least 1). These two instructions are always required by the algorithm, regardless of the value of n. The for loop initialization code also has to always run. This gives us two more instructions; an assignment and a comparison:

i = 0;

i < n;

These will run before the first for loop iteration. After each for loop iteration, we need two more instructions to run, an increment of i and a comparison to check if we'll stay in the loop:

++i;

i < n;

So, if we ignore the loop body, the number of instructions this algorithm needs is 4 + 2n. That is, 4 instructions at the beginning of the for loop and 2 instructions at the end of each iteration of which we have n. We can now define a mathematical function f( n ) that, given an n, gives us the number of instructions the algorithm needs. For an empty for body, we have f( n ) = 4 + 2n.

**Worst-case analysis**

Now, looking at the for body, we have an array lookup operation and a comparison that happen always:

if ( A[ i ] >= M ) { ...

That's two instructions right there. But the if body may run or may not run, depending on what the array values actually are. If it happens to be so that A[ i ] >= M, then we'll run these two additional instructions — an array lookup and an assignment:

M = A[ i ]

But now we can't define an f( n ) as easily, because our number of instructions doesn't depend solely on n but also on our input. For example, for A = [ 1, 2, 3, 4 ] the algorithm will need more instructions than for A = [ 4, 3, 2, 1 ]. When analyzing algorithms, we often consider the worst-case scenario. What's the worst that can happen for our algorithm? When does our algorithm need the most instructions to complete? In this case, it is when we have an array in increasing order such as A = [ 1, 2, 3, 4 ]. In that case, M needs to be replaced every single time and so that yields the most instructions. Computer scientists have a fancy name for that and they call it worst-case analysis; that's nothing more than just considering the case when we're the most unlucky. So, in the worst case, we have 4 instructions to run within the for body, so we have f( n ) = 4 + 2n + 4n = 6n + 4. This function f, given a problem size n, gives us the number of instructions that would be needed in the worst-case.

**Asymptotic behavior**

Given such a function, we have a pretty good idea of how fast an algorithm is. However, as I promised, we won't be needing to go through the tedious task of counting instructions in our program. Besides, the number of actual CPU instructions needed for each programming language statement depends on the compiler of our programming language and on the available CPU instruction set (i.e. whether it's an AMD or an Intel Pentium on your PC, or a MIPS processor on your Playstation 2) and we said we'd be ignoring that. We'll now run our "f" function through a "filter" which will help us get rid of those minor details that computer scientists prefer to ignore.

In our function, 6n + 4, we have two terms: 6n and 4. In complexity analysis we only care about what happens to the instruction-counting function as the program input (n) grows large. This really goes along with the previous ideas of "worst-case scenario" behavior: We're interested in how our algorithm behaves when treated badly; when it's challenged to do something hard. Notice that this is really useful when comparing algorithms. If an algorithm beats another algorithm for a large input, it's most probably true that the faster algorithm remains faster when given an easier, smaller input. **From the terms that we are considering, we'll drop all the terms that grow slowly and only keep the ones that grow fast as n becomes larger.** Clearly 4 remains a 4 as n grows larger, but 6n grows larger and larger, so it tends to matter more and more for larger problems. Therefore, the first thing we will do is drop the 4 and keep the function as f( n ) = 6n.

This makes sense if you think about it, as the 4 is simply an "initialization constant". Different programming languages may require a different time to set up. For example, Java needs some time to initialize its [virtual machine](http://en.wikipedia.org/wiki/Java_virtual_machine). Since we're ignoring programming language differences, it only makes sense to ignore this value.

The second thing we'll ignore is the constant multiplier in front of n, and so our function will become f( n ) = n. As you can see this simplifies things quite a lot. Again, it makes some sense to drop this multiplicative constant if we think about how different programming languages compile. The "array lookup" statement in one language may compile to different instructions in different programming languages. For example, in C, doing A[ i ] does not include a check that i is within the declared array size, while in [Pascal](http://en.wikipedia.org/wiki/Pascal_(programming_language)) it does. So, the following Pascal code:

M := A[ i ]

Is the equivalent of the following in C:

if ( i >= 0 && i < n ) {

M = A[ i ];

}

So it's reasonable to expect that different programming languages will yield different factors when we count their instructions. In our example in which we are using a dumb compiler for Pascal that is oblivious of possible optimizations, Pascal requires 3 instructions for each array access instead of the 1 instruction C requires. Dropping this factor goes along the lines of ignoring the differences between particular programming languages and compilers and only analyzing the idea of the algorithm itself.

This filter of "dropping all factors" and of "keeping the largest growing term" as described above is what we call asymptotic behavior. So the asymptotic behavior of f( n ) = 2n + 8 is described by the function f( n ) = n. Mathematically speaking, what we're saying here is that we're interested in the limit of function f as n tends to infinity; but if you don't understand what that phrase formally means, don't worry, because this is all you need to know. (On a side note, in a strict mathematical setting, we would not be able to drop the constants in the limit; but for computer science purposes, we want to do that for the reasons described above.) Let's work a couple of examples to familiarize ourselves with the concept.

Let us find the asymptotic behavior of the following example functions by dropping the constant factors and by keeping the terms that grow the fastest.

f( n ) = 5n + 12 gives f( n ) = n.

By using the exact same reasoning as above.

f( n ) = 109 gives f( n ) = 1.

We're dropping the multiplier 109 \* 1, but we still have to put a 1 here to indicate that this function has a non-zero value.

f( n ) = n2 + 3n + 112 gives f( n ) = n2

Here, n2 grows larger than 3n for sufficiently large n, so we're keeping that.

f( n ) = n3 + 1999n + 1337 gives f( n ) = n3

Even though the factor in front of n is quite large, we can still find a large enough n so that n3 is bigger than 1999n. As we're interested in the behavior for very large values of n, we only keep n3 (See **Figure 2**).

f( n ) = n + sqrt( n ) gives f( n ) = n

This is so because n grows faster than sqrt( n ) as we increase n.

You can try out the following examples on your own:

**Complexity**

So what this is telling us is that since we can drop all these decorative constants, it's pretty easy to tell the asymptotic behavior of the instruction-counting function of a program. In fact, any program that doesn't have any loops will have f( n ) = 1, since the number of instructions it needs is just a constant (unless it uses recursion; see below). Any program with a single loop which goes from 1 to n will have f( n ) = n, since it will do a constant number of instructions before the loop, a constant number of instructions after the loop, and a constant number of instructions within the loop which all run n times.

This should now be much easier and less tedious than counting individual instructions, so let's take a look at a couple of examples to get familiar with this. The following [PHP](http://php.net/) program checks to see if a particular value exists within an array A of size n:

<?php

$exists = false;

for ( $i = 0; $i < n; ++$i ) {

if ( $A[ $i ] == $value ) {

$exists = true;

break;

}

}

?>

This method of searching for a value within an array is called linear search. This is a reasonable name, as this program has f( n ) = n (we'll define exactly what "linear" means in the next section). You may notice that there's a "break" statement here that may make the program terminate sooner, even after a single iteration. But recall that we're interested in the worst-case scenario, which for this program is for the array A to not contain the value. So we still have f( n ) = n.

Exercise 2

Systematically analyze the number of instructions the above PHP program needs with respect to n in the worst-case to find f( n ), similarly to how we analyzed our first Javascript program. Then verify that, asymptotically, we have f( n ) = n.

Let's look at a Python program which adds two array elements together to produce a sum which it stores in another variable:

v = a[ 0 ] + a[ 1 ]

Here we have a constant number of instructions, so we have f( n ) = 1.

The following program in C++ checks to see if a vector (a fancy array) named A of size n contains the same two values anywhere within it:

bool duplicate = false;

for ( int i = 0; i < n; ++i ) {

for ( int j = 0; j < n; ++j ) {

if ( i != j && A[ i ] == A[ j ] ) {

duplicate = true;

break;

}

}

if ( duplicate ) {

break;

}

}

As here we have two nested loops within each other, we'll have an asymptotic behavior described by f( n ) = n2.

**Rule of thumb**: Simple programs can be analyzed by counting the nested loops of the program. A single loop over n items yields f( n ) = n. A loop within a loop yields f( n ) = n2. A loop within a loop within a loop yields f( n ) = n3.

If we have a program that calls a function within a loop and we know the number of instructions the called function performs, it's easy to determine the number of instructions of the whole program. Indeed, let's take a look at this C example:

int i;

for ( i = 0; i < n; ++i ) {

f( n );

}

If we know that f( n ) is a function that performs exactly n instructions, we can then know that the number of instructions of the whole program is asymptotically n2, as the function is called exactly n times.

**Rule of thumb**: Given a series of for loops that are sequential, the slowest of them determines the asymptotic behavior of the program. Two nested loops followed by a single loop is asymptotically the same as the nested loops alone, because the nested loops dominate the simple loop.

Now, let's switch over to the fancy notation that computer scientists use. When we've figured out the exact such f asymptotically, we'll say that our program is Θ( f( n ) ). For example, the above programs are Θ( 1 ), Θ( n2 ) and Θ( n2 ) respectively. Θ( n ) is pronounced "theta of n". Sometimes we say that f( n ), the original function counting the instructions including the constants, is Θ( something ). For example, we may say that f( n ) = 2n is a function that is Θ( n ) — nothing new here. We can also write 2n ∈ Θ( n ), which is pronounced as "two n is theta of n". Don't get confused about this notation: All it's saying is that if we've counted the number of instructions a program needs and those are 2n, then the asymptotic behavior of our algorithm is described by n, which we found by dropping the constants. Given this notation, the following are some true mathematical statements:

1. n6 + 3n ∈ Θ( n6 )
2. 2n + 12 ∈ Θ( 2n )
3. 3n + 2n ∈ Θ( 3n )
4. nn + n ∈ Θ( nn )

By the way, if you solved Exercise 1 from above, these are exactly the answers you should have found.

**We call this function, i.e. what we put within Θ( here ), the time complexity or just complexity of our algorithm.** So an algorithm with Θ( n ) is of complexity n. We also have special names for Θ( 1 ), Θ( n ), Θ( n2 ) and Θ( log( n ) ) because they occur very often. We say that a Θ( 1 ) algorithm is a constant-time algorithm, Θ( n ) is linear, Θ( n2 ) is quadratic and Θ( log( n ) ) is logarithmic (don't worry if you don't know what logarithms are yet – we'll get to that in a minute).

**Rule of thumb**: Programs with a bigger Θ run slower than programs with a smaller Θ.

**Big-O notation**

Now, it's sometimes true that it will be hard to figure out exactly the behavior of an algorithm in this fashion as we did above, especially for more complex examples. However, we will be able to say that the behavior of our algorithm will never exceed a certain bound. This will make life easier for us, as we won't have to specify exactly how fast our algorithm runs, even when ignoring constants the way we did before. All we'll have to do is find a certain bound. This is explained easily with an example.

A famous problem computer scientists use for teaching algorithms is the sorting problem. In the sorting problem, an array A of size n is given (sounds familiar?) and we are asked to write a program that sorts this array. This problem is interesting because it is a pragmatic problem in real systems. For example, a file explorer needs to sort the files it displays by name so that the user can navigate them with ease. Or, as another example, a video game may need to sort the 3D objects displayed in the world based on their distance from the player's eye inside the virtual world in order to determine what is visible and what isn't, something called the [Visibility Problem](http://en.wikipedia.org/wiki/Hidden_surface_determination) (see **Figure 3**). The objects that turn out to be closest to the player are those visible, while those that are further may get hidden by the objects in front of them. Sorting is also interesting because there are many algorithms to solve it, some of which are worse than others. It's also an easy problem to define and to explain. So let's write a piece of code that sorts an array.

Here is an inefficient way to implement sorting an array in Ruby. (Of course, Ruby supports sorting arrays using build-in functions which you should use instead, and which are certainly faster than what we'll see here. But this is here for illustration purposes.)

b = []

n.times do

m = a[ 0 ]

mi = 0

a.each\_with\_index do |element, i|

if element < m

m = element

mi = i

end

end

a.delete\_at( mi )

b << m

end

This method is called [selection sort](http://en.wikipedia.org/wiki/Selection_sort). It finds the minimum of our array (the array is denoted a above, while the minimum value is denoted m and mi is its index), puts it at the end of a new array (in our case b), and removes it from the original array. Then it finds the minimum between the remaining values of our original array, appends that to our new array so that it now contains two elements, and removes it from our original array. It continues this process until all items have been removed from the original and have been inserted into the new array, which means that the array has been sorted. In this example, we can see that we have two nested loops. The outer loop runs n times, and the inner loop runs once for each element of the array a. While the array a initially has n items, we remove one array item in each iteration. So the inner loop repeats n times during the first iteration of the outer loop, then n - 1 times, then n - 2 times and so forth, until the last iteration of the outer loop during which it only runs once.

It's a little harder to evaluate the complexity of this program, as we'd have to figure out the sum 1 + 2 + ... + (n - 1) + n. But we can for sure find an "upper bound" for it. That is, we can alter our program (you can do that in your mind, not in the actual code) to make it **worse** than it is and then find the complexity of that new program that we derived. If we can find the complexity of the worse program that we've constructed, then we know that our original program is at most that bad, or maybe better. That way, if we find out a pretty good complexity for our altered program, which is worse than our original, we can know that our original program will have a pretty good complexity too – either as good as our altered program or even better.

Let's now think of the way to edit this example program to make it easier to figure out its complexity. But let's keep in mind that we can only make it worse, i.e. make it take up more instructions, so that our estimate is meaningful for our original program. Clearly we can alter the inner loop of the program to always repeat exactly n times instead of a varying number of times. Some of these repetitions will be useless, but it will help us analyze the complexity of the resulting algorithm. If we make this simple change, then the new algorithm that we've constructed is clearly Θ( n2 ), because we have two nested loops where each repeats exactly n times. If that is so, we say that the original algorithm is O( n2 ). O( n2 ) is pronounced "big oh of n squared". What this says is that our program is asymptotically no worse than n2. It may even be better than that, or it may be the same as that. By the way, if our program is indeed Θ( n2 ), we can still say that it's O( n2 ). To help you realize that, imagine altering the original program in a way that doesn't change it much, but still makes it a little worse, such as adding a meaningless instruction at the beginning of the program. Doing this will alter the instruction-counting function by a simple constant, which is ignored when it comes to asymptotic behavior. So a program that is Θ( n2 ) is also O( n2 ).

But a program that is O( n2 ) may not be Θ( n2 ). For example, any program that is Θ( n ) is also O( n2 ) in addition to being O( n ). If we imagine the that a Θ( n ) program is a simple forloop that repeats n times, we can make it worse by wrapping it in another for loop which repeats n times as well, thus producing a program with f( n ) = n2. To generalize this, any program that is Θ( a ) is O( b ) when b is worse than a. Notice that our alteration to the program doesn't need to give us a program that is actually meaningful or equivalent to our original program. It only needs to perform more instructions than the original for a given n. All we're using it for is counting instructions, not actually solving our problem.

So, saying that our program is O( n2 ) is being on the safe side: We've analyzed our algorithm, and we've found that it's never worse than n2. But it could be that it's in fact n2. This gives us a good estimate of how fast our program runs. Let's go through a few examples to help you familiarize yourself with this new notation.

You may be getting a little overwhelmed with all this new notation by now, but let's introduce just two more symbols before we move on to a few examples. These are easy now that you know Θ, O and o, and we won't use them much later in this article, but it's good to know them now that we're at it. In the example above, we modified our program to make it worse (i.e. taking more instructions and therefore more time) and created the O notation. O is meaningful because it tells us that our program will never be slower than a specific bound, and so it provides valuable information so that we can argue that our program is good enough. If we do the opposite and modify our program to make it **better** and find out the complexity of the resulting program, we use the notation Ω. Ω therefore gives us a complexity that we know our program won't be better than. This is useful if we want to prove that a program runs slowly or an algorithm is a bad one. This can be useful to argue that an algorithm is too slow to use in a particular case. For example, saying that an algorithm is Ω( n3 ) means that the algorithm isn't better than n3. It might be Θ( n3 ), as bad as Θ( n4 ) or even worse, but we know it's at least somewhat bad. So Ω gives us a lower bound for the complexity of our algorithm. Similarly to ο, we can write ω if we know that our bound isn't tight. For example, a Θ( n3 ) algorithm is ο( n4 ) and ω( n2 ). Ω( n ) is pronounced "big omega of n", while ω( n ) is pronounced "small omega of n".

The reason we use O and Ω instead of Θ even though O and Ω can also give tight bounds is that we may not be able to tell if a bound we've found is tight, or we may just not want to go through the process of scrutinizing it so much.

If you don't fully remember all the different symbols and their uses, don't worry about it too much right now. You can always come back and look them up. The most important symbols are O and Θ.

Also note that although Ω gives us a lower-bound behavior for our function (i.e. we've improved our program and made it perform less instructions) we're still referring to a "worst-case" analysis. This is because we're feeding our program the worst possible input for a given n and analyzing its behavior under this assumption.

The following table indicates the symbols we just introduced and their correspondence with the usual mathematical symbols of comparisons that we use for numbers. The reason we don't use the usual symbols here and use Greek letters instead is to point out that we're doing an asymptotic behavior comparison, not just a simple comparison.

| **Asymptotic comparison operator** | **Numeric comparison operator** |
| --- | --- |
| Our algorithm is **o**( something ) | A number is **<** something |
| Our algorithm is **O**( something ) | A number is **≤** something |
| Our algorithm is **Θ**( something ) | A number is **=** something |
| Our algorithm is **Ω**( something ) | A number is **≥** something |
| Our algorithm is **ω**( something ) | A number is **>** something |

**Rule of thumb**: While all the symbols O, o, Ω, ω and Θ are useful at times, O is the one used more commonly, as it's easier to determine than Θ and more practically useful than Ω.

Logarithms

If you know what logarithms are, feel free to skip this section. As a lot of people are unfamiliar with logarithms, or just haven't used them much recently and don't remember them, this section is here as an introduction for them. This text is also for younger students that haven't seen logarithms at school yet. Logarithms are important because they occur a lot when analyzing complexity. A logarithm is an operation applied to a number that makes it quite smaller – much like a square root of a number. So if there's one thing you want to remember about logarithms is that they take a number and make it much smaller than the original (See **Figure 4**). Now, in the same way that square roots are the inverse operation of squaring something, logarithms are the inverse operation of exponentiating something. This isn't as hard as it sounds. It's better explained with an example. Consider the equation:

2x = 1024

We now wish to solve this equation for x. So we ask ourselves: What is the number to which we must raise the base 2 so that we get 1024? That number is 10. Indeed, we have 210 = 1024, which is easy to verify. Logarithms help us denote this problem using new notation. In this case, 10 is the logarithm of 1024 and we write this as log( 1024 ) and we read it as "the logarithm of 1024". Because we're using 2 as a base, these logarithms are called base 2 logarithms. There are logarithms in other bases, but we'll only use base 2 logarithms in this article. If you're a student competing in international competitions and you don't know about logarithms, I highly recommend that you [practice your logarithms](http://tutorial.math.lamar.edu/Classes/Alg/LogFunctions.aspx) after completing this article. In computer science, base 2 logarithms are much more common than any other types of logarithms. This is because we often only have two different entities: 0 and 1. We also tend to cut down one big problem into halves, of which there are always two. So you only need to know about base-2 logarithms to continue with this article.

**Rule of thumb**: For competition algorithms implemented in C++, once you've analyzed your complexity, you can get a rough estimate of how fast your program will run by expecting it to perform about 1,000,000 operations per second, where the operations you count are given by the asymptotic behavior function describing your algorithm. For example, a Θ( n ) algorithm takes about a second to process the input for n = 1,000,000.

Recursive complexity

Let's now take a look at a recursive function. A recursive function is a function that calls itself. Can we analyze its complexity? The following function, written in Python, evaluates the [factorial](http://en.wikipedia.org/wiki/Factorial) of a given number. The factorial of a positive integer number is found by multiplying it with all the previous positive integers together. For example, the factorial of 5 is 5 \* 4 \* 3 \* 2 \* 1. We denote that "5!" and pronounce it "five factorial" (some people prefer to pronounce it by screaming it out aloud like "FIVE!!!")

def factorial( n ):

if n == 1:

return 1

return n \* factorial( n - 1 )

Let us analyze the complexity of this function. This function doesn't have any loops in it, but its complexity isn't constant either. What we need to do to find out its complexity is again to go about counting instructions. Clearly, if we pass some n to this function, it will execute itself n times. If you're unsure about that, run it "by hand" now for n = 5 to validate that it actually works. For example, for n = 5, it will execute 5 times, as it will keep decreasing n by 1 in each call. We can see therefore that this function is then Θ( n ).

If you're unsure about this fact, remember that you can always find the exact complexity by counting instructions. If you wish, you can now try to count the actual instructions performed by this function to find a function f( n ) and see that it's indeed linear (recall that linear means Θ( n )).

See **Figure 5** for a diagram to help you understand the recursions performed when factorial( 5 ) is called.

This should clear up why this function is of linear complexity.

**Logarithmic complexity**

One famous problem in computer science is that of searching for a value within an array. We solved this problem earlier for the general case. This problem becomes interesting if we have an array which is sorted and we want to find a given value within it. One method to do that is called binary search. We look at the middle element of our array: If we find it there, we're done. Otherwise, if the value we find there is bigger than the value we're looking for, we know that our element will be on the left part of the array. Otherwise, we know it'll be on the right part of the array. We can keep cutting these smaller arrays in halves until we have a single element to look at. Here's the method using pseudocode:

def binarySearch( A, n, value ):

if n = 1:

if A[ 0 ] = value:

return true

else:

return false

if value < A[ n / 2 ]:

return binarySearch( A[ 0...( n / 2 - 1 ) ], n / 2 - 1, value )

else if value > A[ n / 2 ]:

return binarySearch( A[ ( n / 2 + 1 )...n ], n / 2 - 1, value )

else:

return true

This pseudocode is a simplification of the actual implementation. In practice, this method is easier described than implemented, as the programmer needs to take care of some implementation issues. There are off-by-one errors and the division by 2 may not always produce an integer value and so it's necessary to floor() or ceil() the value. But we can assume for our purposes that it will always succeed, and we'll assume our actual implementation in fact takes care of the off-by-one errors, as we only want to analyze the complexity of this method. If you've never implemented binary search before, you may want to do this in your favourite programming language. It's a truly enlightening endeavor.

See **Figure 6** to help you understand the way binary search operates.

If you're unsure that this method actually works, take a moment now to run it by hand in a simple example and convince yourself that it actually works.

Let us now attempt to analyze this algorithm. Again, we have a recursive algorithm in this case. Let's assume, for simplicity, that the array is always cut in exactly a half, ignoring just now the + 1 and - 1 part in the recursive call. By now you should be convinced that a little change such as ignoring + 1 and - 1 won't affect our complexity results. This is a fact that we would normally have to prove if we wanted to be prudent from a mathematical point of view, but practically it is intuitively obvious. Let's assume that our array has a size that is an exact power of 2, for simplicity. Again this assumption doesn't change the final results of our complexity that we will arrive at. The worst-case scenario for this problem would happen when the value we're looking for does not occur in our array at all. In that case, we'd start with an array of size n in the first call of the recursion, then get an array of size n / 2 in the next call. Then we'll get an array of size n / 4 in the next recursive call, followed by an array of size n / 8 and so forth. In general, our array is split in half in every call, until we reach 1. So, let's write the number of elements in our array for every call:

0th iteration: n

1st iteration: n / 2

2nd iteration: n / 4

3rd iteration: n / 8

...

ith iteration: n / 2i

...

last iteration: 1

Notice that in the i-th iteration, our array has n / 2i elements. This is because in every iteration we're cutting our array into half, meaning we're dividing its number of elements by two. This translates to multiplying the denominator with a 2. If we do that i times, we get n / 2i. Now, this procedure continues and with every larger i we get a smaller number of elements until we reach the last iteration in which we have only 1 element left. If we wish to find i to see in what iteration this will take place, we have to solve the following equation:

1 = n / 2i

This will only be true when we have reached the final call to the binarySearch() function, not in the general case. So solving for i here will help us find in which iteration the recursion will finish. Multiplying both sides by 2i we get:

2i = n

Now, this equation should look familiar if you read the logarithms section above. Solving for i we have:

i = log( n )

This tells us that the number of iterations required to perform a binary search is log( n ) where n is the number of elements in the original array.

If you think about it, this makes some sense. For example, take n = 32, an array of 32 elements. How many times do we have to cut this in half to get only 1 element? We get: 32 → 16 → 8 → 4 → 2 → 1. We did this 5 times, which is the logarithm of 32. Therefore, the complexity of binary search is Θ( log( n ) ).

This last result allows us to compare binary search with linear search, our previous method. Clearly, as log( n ) is much smaller than n, it is reasonable to conclude that binary search is a much faster method to search within an array then linear search, so it may be advisable to keep our arrays sorted if we want to do many searches within them.

**Rule of thumb**: Improving the asymptotic running time of a program often tremendously increases its performance, much more than any smaller "technical" optimizations such as using a faster programming language.

Optimal sorting

**Congratulations.** You now know about analyzing the complexity of algorithms, asymptotic behavior of functions and big-O notation. You also know how to intuitively figure out that the complexity of an algorithm is O( 1 ), O( log( n ) ), O( n ), O( n2 ) and so forth. You know the symbols o, O, ω, Ω and Θ and what worst-case analysis means. If you've come this far, this tutorial has already served its purpose.

This final section is optional. It is a little more involved, so feel free to skip it if you feel overwhelmed by it. It will require you to focus and spend some moments working through the exercises. However, it will provide you with a very useful method in algorithm complexity analysis which can be very powerful, so it's certainly worth understanding.

We looked at a sorting implementation above called a selection sort. We mentioned that selection sort is not optimal. An optimal algorithm is an algorithm that solves a problem in the best possible way, meaning there are no better algorithms for this. This means that all other algorithms for solving the problem have a worse or equal complexity to that optimal algorithm. There may be many optimal algorithms for a problem that all share the same complexity. The sorting problem can be solved optimally in various ways. We can use the same idea as with binary search to sort quickly. This sorting method is called mergesort.

To perform a mergesort, we will first need to build a helper function that we will then use to do the actual sorting. We will make a merge function which takes two arrays that are both already sorted and merges them together into a big sorted array. This is easily done:

def merge( A, B ):

if empty( A ):

return B

if empty( B ):

return A

if A[ 0 ] < B[ 0 ]:

return concat( A[ 0 ], merge( A[ 1...A\_n ], B ) )

else:

return concat( B[ 0 ], merge( A, B[ 1...B\_n ] ) )

The concat function takes an item, the "head", and an array, the "tail", and builds up and returns a new array which contains the given "head" item as the first thing in the new array and the given "tail" item as the rest of the elements in the array. For example, concat( 3, [ 4, 5, 6 ] ) returns [ 3, 4, 5, 6 ]. We use A\_n and B\_n to denote the sizes of arrays A and B respectively.

Let's see what's going on here. Each circle represents a call to the mergeSort function. The number written in the circle indicates the size of the array that is being sorted. The top blue circle is the original call to mergeSort, where we get to sort an array of size n. The arrows indicate recursive calls made between functions. The original call to mergeSort makes two calls to mergeSort on two arrays, each of size n / 2. This is indicated by the two arrows at the top. In turn, each of these calls makes two calls of its own to mergeSort two arrays of size n / 4 each, and so forth until we arrive at arrays of size 1. This diagram is called a recursion tree, because it illustrates how the recursion behaves and looks like a tree (the root is at the top and the leaves are at the bottom, so in reality it looks like an inversed tree).

Notice that at each row in the above diagram, the total number of elements is n. To see this, take a look at each row individually. The first row contains only one call to mergeSort with an array of size n, so the total number of elements is n. The second row has two calls to mergeSort each of size n / 2. But n / 2 + n / 2 = n and so again in this row the total number of elements is n. In the third row, we have 4 calls each of which is applied on an n / 4-sized array, yielding a total number of elements equal to n / 4 + n / 4 + n / 4 + n / 4 = 4n / 4 = n. So again we get n elements. Now notice that at each row in this diagram the caller will have to perform a mergeoperation on the elements returned by the callees. For example, the circle indicated with red color has to sort n / 2 elements. To do this, it splits the n / 2-sized array into two n / 4-sized arrays, calls mergeSort recursively to sort those (these calls are the circles indicated with green color), then merges them together. This merge operation requires to merge n / 2 elements. At each row in our tree, the total number of elements merged is n. In the row that we just explored, our function merges n / 2 elements and the function on its right (which is in blue color) also has to merge n / 2 elements of its own. That yields n elements in total that need to be merged for the row we're looking at.

By this argument, the complexity for each row is Θ( n ). We know that the number of rows in this diagram, also called the depth of the recursion tree, will be log( n ). The reasoning for this is exactly the same as the one we used when analyzing the complexity of binary search. We have log( n ) rows and each of them is Θ( n ), therefore the complexity of mergeSort is Θ( n \* log( n ) ). This is much better than Θ( n2 ) which is what selection sort gave us (remember that log( n ) is much smaller than n, and so n \* log( n ) is much smaller than n \* n = n2). If this sounds complicated to you, don't worry: It's not easy the first time you see it. Revisit this section and reread about the arguments here after you implement mergesort in your favourite programming language and validate that it works.

As you saw in this last example, complexity analysis allows us to compare algorithms to see which one is better. Under these circumstances, we can now be pretty certain that merge sort will outperform selection sort for large arrays. This conclusion would be hard to draw if we didn't have the theoretical background of algorithm analysis that we developed. In practice, indeed sorting algorithms of running time Θ( n \* log( n ) ) are used. For example, [the Linux kernel uses a sorting algorithm called heapsort](http://lxr.free-electrons.com/source/lib/sort.c), which has the same running time as mergesort which we explored here, namely Θ( n log( n ) ) and so is optimal. Notice that we have not proven that these sorting algorithms are optimal. Doing this requires a slightly more involved mathematical argument, but rest assured that they can't get any better from a complexity point of view.

Having finished reading this tutorial, the intuition you developed for algorithm complexity analysis should be able to help you design faster programs and focus your optimization efforts on the things that really matter instead of the minor things that don't matter, letting you work more productively. In addition, the mathematical language and notation developed in this article such as big-O notation is helpful in communicating with other software engineers when you want to argue about the running time of algorithms, so hopefully you will be able to do that with your newly acquired knowledge.

<http://stackoverflow.com/questions/487258/plain-english-explanation-of-big-o>

The simplest definition I can give for Big-O notation is this:

**Big-O notation is a relative representation of the complexity of an algorithm.**

There are some important and deliberately chosen words in that sentence:

**relative:** you can only compare apples to apples. You can't compare an algorithm to do arithmetic multiplication to an algorithm that sorts a list of integers. But a comparison of two algorithms to do arithmetic operations (one multiplication, one addition) will tell you something meaningful;

**representation:** Big-O (in its simplest form) reduces the comparison between algorithms to a single variable. That variable is chosen based on observations or assumptions. For example, sorting algorithms are typically compared based on comparison operations (comparing two nodes to determine their relative ordering). This assumes that comparison is expensive. But what if comparison is cheap but swapping is expensive? It changes the comparison; and

**complexity:** if it takes me one second to sort 10,000 elements how long will it take me to sort one million? Complexity in this instance is a relative measure to something else.

Come back and reread the above when you've read the rest.

The best example of Big-O I can think of is doing arithmetic. Take two numbers (123456 and 789012). The basic arithmetic operations we learnt in school were:

addition;

subtraction;

multiplication; and

division.

Each of these is an operation or a problem. A method of solving these is called an **algorithm**.

Addition is the simplest. You line the numbers up (to the right) and add the digits in a column writing the last number of that addition in the result. The 'tens' part of that number is carried over to the next column.

Let's assume that the addition of these numbers is the most expensive operation in this algorithm. It stands to reason that to add these two numbers together we have to add together 6 digits (and possibly carry a 7th). If we add two 100 digit numbers together we have to do 100 additions. If we add two 10,000 digit numbers we have to do 10,000 additions.

See the pattern? The **complexity** (being the number of operations) is directly proportional to the number of digits *n* in the larger number. We call this **O(n)** or **linear complexity**.

Subtraction is similar (except you may need to borrow instead of carry).

Multiplication is different. You line the numbers up, take the first digit in the bottom number and multiply it in turn against each digit in the top number and so on through each digit. So to multiply our two 6 digit numbers we must do 36 multiplications. We may need to do as many as 10 or 11 column adds to get the end result too.

If we have two 100-digit numbers we need to do 10,000 multiplications and 200 adds. For two one million digit numbers we need to do one trillion (1012) multiplications and two million adds.

As the algorithm scales with n-*squared*, this is **O(n2)** or **quadratic complexity**. This is a good time to introduce another important concept:

**We only care about the most significant portion of complexity.**

The astute may have realized that we could express the number of operations as: n2 + 2n. But as you saw from our example with two numbers of a million digits apiece, the second term (2n) becomes insignificant (accounting for 0.0002% of the total operations by that stage).

One can notice that we've assumed the worst case scenario here. While multiplying 6 digit numbers if one of them is 4 digit and the other one is 6 digit, then we only have 24 multiplications. Still we calculate the worst case scenario for that 'n', i.e when both are 6 digit numbers. Hence Big-O notation is about the Worst-case scenario of an algorithm

**The Telephone Book**

The next best example I can think of is the telephone book, normally called the White Pages or similar but it'll vary from country to country. But I'm talking about the one that lists people by surname and then initials or first name, possibly address and then telephone numbers.

Now if you were instructing a computer to look up the phone number for "John Smith" in a telephone book that contains 1,000,000 names, what would you do? Ignoring the fact that you could guess how far in the S's started (let's assume you can't), what would you do?

A typical implementation might be to open up to the middle, take the 500,000th and compare it to "Smith". If it happens to be "Smith, John", we just got real lucky. Far more likely is that "John Smith" will be before or after that name. If it's after we then divide the last half of the phone book in half and repeat. If it's before then we divide the first half of the phone book in half and repeat. And so on.

This is called a **binary search** and is used every day in programming whether you realize it or not.

So if you want to find a name in a phone book of a million names you can actually find any name by doing this at most 20 times. In comparing search algorithms we decide that this comparison is our 'n'.

For a phone book of 3 names it takes 2 comparisons (at most).

For 7 it takes at most 3.

For 15 it takes 4.

…

For 1,000,000 it takes 20.

That is staggeringly good isn't it?

In Big-O terms this is **O(log n)** or **logarithmic complexity**. Now the logarithm in question could be ln (base e), log10, log2 or some other base. It doesn't matter it's still O(log n) just like O(2n2) and O(100n2) are still both O(n2).

It's worthwhile at this point to explain that Big O can be used to determine three cases with an algorithm:

**Best Case:** In the telephone book search, the best case is that we find the name in one comparison. This is **O(1)** or **constant complexity**;

**Expected Case:** As discussed above this is O(log n); and

**Worst Case:** This is also O(log n).

Normally we don't care about the best case. We're interested in the expected and worst case. Sometimes one or the other of these will be more important.

Back to the telephone book.

What if you have a phone number and want to find a name? The police have a reverse phone book but such look-ups are denied to the general public. Or are they? Technically you can reverse look-up a number in an ordinary phone book. How?

You start at the first name and compare the number. If it's a match, great, if not, you move on to the next. You have to do it this way because the phone book is **unordered** (by phone number anyway).

So to find a name:

**Best Case:** O(1);

**Expected Case:** O(n) (for 500,000); and

**Worst Case:** O(n) (for 1,000,000).

**The Travelling Salesman**

This is quite a famous problem in computer science and deserves a mention. In this problem you have N towns. Each of those towns is linked to 1 or more other towns by a road of a certain distance. The Travelling Salesman problem is to find the shortest tour that visits every town.

Sounds simple? Think again.

If you have 3 towns A, B and C with roads between all pairs then you could go:

* A → B → C
* A → C → B
* B → C → A
* B → A → C
* C → A → B
* C → B → A

Well actually there's less than that because some of these are equivalent (A → B → C and C → B → A are equivalent, for example, because they use the same roads, just in reverse).

In actuality there are 3 possibilities.

Take this to 4 towns and you have (iirc) 12 possibilities.

With 5 it's 60.

6 becomes 360.

This is a function of a mathematical operation called a **factorial**. Basically:

5! = 5 × 4 × 3 × 2 × 1 = 120

6! = 6 × 5 × 4 × 3 × 2 × 1 = 720

7! = 7 × 6 × 5 × 4 × 3 × 2 × 1 = 5040

…

25! = 25 × 24 × … × 2 × 1 = 15,511,210,043,330,985,984,000,000

…

50! = 50 × 49 × … × 2 × 1 = 3.04140932 × 1064

So the Big-O of the Travelling Salesman problem is **O(n!)** or **factorial or combinatorial complexity**.

**By the time you get to 200 towns there isn't enough time left in the universe to solve the problem with traditional computers.**

Something to think about.

**Polynomial Time**

Another point I wanted to make quick mention of is that any algorithm that has a complexity of O(na) is said to have polynomial complexity or is solvable in polynomial time.

Traditional computers can solve polynomial-time problems. Certain things are used in the world because of this. Public Key Cryptography is a prime example. It is computationally hard to find two prime factors of a very large number. If it wasn't, we couldn't use the public key systems we use.

Anyway, that's it for my (hopefully plain English) explanation of Big O (revised).

It shows how an algorithm scales.

**O(n2):**

* 1 item: 1 second
* 10 items: 100 seconds
* 100 items: 10000 seconds

Notice that the number of items increases by a factor of 10, but the time increases by a factor of 102. Basically, n=10 and so O(n2) gives us the scaling factor n2 which is 102.

**O(n):**

* 1 item: 1 second
* 10 items: 10 seconds
* 100 items: 100 seconds

This time the number of items increases by a factor of 10, and so does the time. n=10 and so O(n)'s scaling factor is 10.

**O(1):**

* 1 item: 1 second
* 10 items: 1 second
* 100 items: 1 second

The number of items is still increasing by a factor of 10, but the scaling factor of O(1) is always 1.

That's the gist of it. They reduce the maths down so it might not be exactly n2 or whatever they say it is, but that'll be the dominating factor in the scaling.

This is almost certainly confusing [Big O notation](http://en.wikipedia.org/wiki/Big_O_notation) (which is an upper bound) with Theta notation (which is both an upper and lower bound). In my experience this is actually typical of discussions in non-academic settings. Apologies for any confusion caused.

In one sentence: As the size of your job goes up, how much longer does it take to complete it?

Obviously that's only using "size" as the input and "time taken" as the output — the same idea applies if you want to talk about memory usage etc.

Here's an example where we have N T-shirts which we want to dry. We'll assume it's incredibly quick to get them in the drying position (i.e. the human interaction is negligible). That's not the case in real life, of course...

Using a washing line outside: assuming you have an infinitely large back yard, washing dries in O(1) time. However much you have of it, it'll get the same sun and fresh air, so the size doesn't affect the drying time.

Using a tumble dryer: you put 10 shirts in each load, and then they're done an hour later. (Ignore the actual numbers here — they're irrelevant.) So drying 50 shirts takes about 5 times as long as drying 10 shirts.

Putting everything in an airing cupboard: If we put everything in one big pile and just let general warmth do it, it will take a long time for the middle shirts to get dry. I wouldn't like to guess at the detail, but I suspect this is at least O(N^2) — as you increase the wash load, the drying time increases faster.

One important aspect of "big O" notation is that it doesn't say which algorithm will be faster for a given size. Take a hashtable (string key, integer value) vs an array of pairs (string, integer). Is it faster to find a key in the hashtable or an element in the array, based on a string? (i.e. for the array, "find the first element where the string part matches the given key.") Hashtables are generally amortised (~= "on average") O(1) — once they're set up, it should take about the same time to find an entry in a 100 entry table as in a 1,000,000 entry table. Finding an element in an array (based on content rather than index) is linear, i.e. O(N) — on average, you're going to have to look at half the entries.

Does this make a hashtable faster than an array for lookups? Not necessarily. If you've got a very small collection of entries, an array may well be faster — you may be able to check all the strings in the time that it takes to just calculate the hashcode of the one you're looking at. As the data set grows larger, however, the hashtable will eventually beat the array.

**Basics**

**for "sufficiently" large inputs...**

f(x) ∈ O(upperbound) means f "grows no faster than" upperbound

f(x) ∈ Ɵ(justlikethis) mean f "grows exactly like" justlikethis

f(x) ∈ Ω(lowerbound) means f "grows no slower than" lowerbound

big-O notation doesn't care about constant factors: the function 9x² is said to "grow exactly like"10x². Neither does big-O *asymptotic* notation care about *non-asymptotic* stuff ("stuff near the origin" or "what happens when the problem size is small"): the function 10x² is said to "grow exactly like" 10x² - x + 2.

Why would you want to ignore the smaller parts of the equation? Because they become completely dwarfed by the big parts of the equation as you consider larger and larger scales; their contribution becomes dwarfed and irrelevant. (See example section.)

Put another way, it's all about the **ratio**. *If you divide the actual time it takes by the O(...), you will get a constant factor in the limit of large inputs.* Intuitively this makes sense: functions "scale like" one another if you can multiply one to get the other. That is, when we say...

actualAlgorithmTime(N) ∈ O(bound(N))

e.g. "time to mergesort N elements

is O(N log(N))"

... this means that ***for "large enough" problem sizes N*** (if we ignore stuff near the origin), there exists some constant (e.g. 2.5, completely made up) such that:

actualAlgorithmTime(N) e.g. "mergesort\_duration(N) "

────────────────────── < constant ───────────────────── < 2.5

bound(N) N log(N)

There are many choices of constant; often the "best" choice is known as the "constant factor" of the algorithm... but we often ignore it like we ignore non-largest terms (see Constant Factors section for why they don't usually matter). You can also think of the above equation as a bound, saying "*In the worst-case scenario, the time it takes will never be worse than roughly N\*log(N), within a factor of 2.5 (a constant factor we don't care much about)*".

In general, O(...) is the most useful one because we often care about worst-case behavior. If f(x)represents something "bad" like processor or memory usage, then "f(x) ∈ O(upperbound)" means "upperbound is the worse-case scenario of processor/memory usage".

**Intuition**

This lets us make statements like...

"For large enough inputsize=N, and a constant

factor of 1, if I double the input size...

... I double the time it takes." ( O(N) )

... I quadruple the time it takes." ( O(N²) )

... I add 1 to the time it takes." ( O(log(N)) )

... I don't change the time it takes." ( O(1) )

Big O describes an upper limit on the growth behaviour of a function, for example the runtime of a program, when inputs become large.

Examples:

O(n): If I double the input size the runtime doubles

O(n2): If the input size doubles the runtime quadruples

O(log n): If the input size doubles the runtime increases by one

O(2n): If the input size increases by one, the runtime doubles

The input size is usually the space in bits needed to represent the input.

**Applications**

As a purely mathematical construct, big-O notation is not limited to talking about processing time and memory. You can use it to discuss the asymptotics of anything where scaling is meaningful, such as:

* the number of possibly handshakes among N people at a party (Ɵ(N²), specifically N(N-1)/2, but what matters is that it "scales like" N²)
* probabilistic expected number of people who have seen some viral marketing as a function of time
* how website latency scales with the number of processing units in a CPU or GPU or computer cluster
* how heat output scales on CPU dies as a function of transistor count, voltage, etc.

**Example**

For the handshake example, #handshakes ∈ Ɵ(N²). The number of handshakes is exactly n-choose-2 or (N²-N)/2 (each of N people shakes the hands of N-1 other people, but this double-counts handshakes so divide by 2). However, for very large numbers of people, the linear term N is dwarfed and effectively contributes 0 to the ratio. Therefore the scaling behavior is order N², or the number of handshakes "grows like N²".

#handshakes(N)

────────────── ≈ 1/2

N²

If you wanted to prove this to yourself, you could perform some simple algebra on the ratio to split it up into multiple terms (lim means "considered in the limit of", you can ignore it if it makes you feel better):

N²/2 - N/2 (N²)/2 N/2 1/2

lim ────────── = lim ( ────── - ─── ) = lim ─── = 1/2

N→∞ N² N→∞ N² N² N→∞ 1

┕━━━┙

this is 0 in the limit of N→∞:

graph it, or plug in a really large number for N

**Constant factors**

Usually we don't care what the specific constant factors are, because they don't affect the way the function grows. For example, two algorithm may both take O(N) time to complete, but one may be twice as slow as the other. We usually don't care too much unless the factor is very large, since optimizing is tricky business ( [When is optimisation premature?](http://stackoverflow.com/questions/385506/when-is-optimisation-premature) ); also the mere act of picking an algorithm with a better big-O will often improve performance by orders of magnitude.

Some asymptotically superior algorithms (e.g. a non-comparison O(N log(log(N))) sort) can have so large a constant factor (e.g. 100000\*N log(log(N))), or overhead that is relatively large like O(N log(log(N))) with a hidden + 100\*N, that they are rarely worth using even on "big data".

**Why O(N) is sometimes the best you can do, i.e. why we need datastructures**

O(N) algorithms are in some sense the "best" algorithms if you need to read all your data. The **very act of reading** a bunch of data is an O(N) operation. Loading it into memory is usually O(N) (or faster if you have hardware support, or no time at all if you've already read the data). However if you touch or even look at every piece of data (or even every other piece of data), your algorithm will takeO(N) time to perform this looking. Nomatter how long your actual algorithm takes, it will be at leastO(N) because it spent that time looking at all the data.

The same can be said for the **very act of writing**. For example, all algorithms which print out all permutations of a number N are O(N!) because the output is at least that long.

This motivates the use of **data structures**: a data structure requires reading the data only once (usuallyO(N) time), plus some arbitrary amount of preprocessing (e.g. O(N) or O(N log(N)) or O(N²)) which we try to keep small. Thereafter, modifying the data structure (insertions / deletions / etc.) and making queries on the data take very little time, such as O(1) or O(log(N)). You then proceed to make a large number of queries! In general, the more work you're willing to do ahead of time, the less work you'll have to do later on.

For example, say you had the latitude and longitude coordinates of millions of roads segments, and wanted to find all street intersections.

* Naive method: If you had the coordinates of a street intersection, and wanted to examine nearby streets, you would have to go through the millions of segments each time, and check each one for adjacency.
* If you only needed to do this once, it would not be a problem to have to do the naive method ofO(N) work only once, but if you want to do it many times (in this case, N times, once for each segment), we'd have to do O(N²) work, or 1000000²=1000000000000 operations. Not good (a modern computer can perform about a billion operations per second).
* If we use a simple structure called a hash table (an instant-speed lookup table, also known as a hashmap or dictionary), we pay a small cost by preprocessing everything in O(N) time. Thereafter, it only takes constant time on average to look up something by its key (in this case, our key is the latitude and longitude coordinates, rounded into a grid; we search the adjacent gridspaces of which there are only 9, which is a constant).
* Our task went from an infeasible O(N²) to a manageable O(N), and all we had to do was pay a minor cost to make a hash table.

The moral of the story: a data structure lets us speed up operations. Even more advanced data structures can let you combine, delay, or even ignore operations in incredibly clever ways, like leaving the equivalent of "to-do" notes at junctions in a tree.

**Amortized / average-case complexity**

There is also the concept of "amortized" or "average case". This is no more than using big-O notation for the expected value of a function, rather than the function itself. For example, some data structures may have a worse-case complexity of O(N) for a single operation, but guarantee that if you do many of these operations, the average-case complexity will be O(1).

**Multidimensional big-O**

Most of the time, people don't realize that there's more than one variable at work. For example, in a string-search algorithm, your algorithm may take time O([length of text] + [length of query]), i.e. it is linear in two variables like O(N+M). Other more naive algorithms may be O([length of text]\*[length of query]) or O(N\*M). Ignoring multiple variables is one of the most common oversights I see in algorithm analysis, and can handicap you when designing an algorithm.

**The whole story**

Keep in mind that big-O is not the whole story. You can drastically speed up some algorithms by using caching, making them cache-oblivious, avoiding bottlenecks by working with RAM instead of disk, using parallelization, or doing work ahead of time -- these techniques are often independent of the order-of-growth "big-O" notation, though you will often see the number of cores in the big-O notation of parallel algorithms.

Also keep in mind that due to hidden constraints of your program, you might not really care about asymptotic behavior. You may be working with a bounded number of values, for example:

* If you're sorting something like 5 elements, you don't want to use the speedy O(N log(N))quicksort; you want to use insertion sort, which happens to perform well on small inputs. These situations often comes up in divide-and-conquer algorithms, where you split up the problem into smaller and smaller subproblems, such as recursive sorting, fast Fourier transforms, or matrix multiplication.
* If some values are effectively bounded due to some hidden fact (e.g. the average human name is softly bounded at perhaps 40 letters, and human age is softly bounded at around 150). You can also impose bounds on your input to effectively make terms constant.

In practice, even among algorithms which have the same or similar asymptotic performance, their relative merit may actually be driven by other things, such as: other performance factors (quicksort and mergesort are both O(N log(N)), but quicksort takes advantage of CPU caches); non-performance considerations, like ease of implementation; whether a library is available, and how reputable and maintained the library is.

Many things can implicitly contribute to the running time's constant factor, such as whether you run your algorithm on a 500MHz computer vs 2GHz computer, whether your programming language is interpreted or using a JIT compiler, whether you are doing a constant amount of extra work in a critical section of code, etc. The effect may be small (e.g. 0.9x speed) or large (e.g. 0.01x speed) compared to a different implementation and/or environment. Do you switch languages to eek out that little extra constant factor of work? That literally depends on a hundred other reasons (necessity, skills, coworkers, programmer productivity, the monetary value of your time, familiarity, workarounds, why not assembly or GPU, etc...), which may be more important than performance.

The above issues, like programming language, are almost never considered as part of the constant factor (nor should they be); yet one should be aware of them, because sometimes (though rarely) they may not be constant. For example in cpython, the native priority queue implementation is asymptotically non-optimal (O(log(N)) rather than O(1) for your choice of insertion or find-min); do you use another implementation? Probably not, since the C implementation is probably faster, and there are probably other similar issues elsewhere. There are tradeoffs; sometimes they matter and sometimes they don't.

Math addenda

*For completeness, the precise definition of big-O notation is as follows: f(x) ∈ O(g(x)) means that "f is asymptotically upper-bounded by const\*g": ignoring everything below some finite value of x, there exists a constant such that |f(x)| ≤ const \* |g(x)|. (The other symbols are as follows: just likeO means ≤, Ω means ≥. There are lowercase variants: o means <, and ω means >.) f(x) ∈ Ɵ(g(x)) means both f(x) ∈ O(g(x)) and f(x) ∈ Ω(g(x)) (upper- and lower-bounded by g): there exists some constants such that f will always lie in the "band" between const1\*g(x) andconst2\*g(x). It is the strongest asymptotic statement you can make and roughly equivalent to ==. (Sorry, I elected to delay the mention of the absolute-value symbols until now, for clarity's sake; especially because I have never seen negative values come up in a computer science context.)*

*People will often use = O(...). It is technically more correct to use ∈ O(...). ∈ means "is an element of". O(N²) is actually an*equivalence class*, that is, it is a set of things which we consider to be the same. In this particular case, O(N²) contains elements like {2 N², 3 N², 1/2 N², 2 N² + log(N), - N² + N^1.9, ...} and is infinitely large, but it's still a set. People will know what you mean if you use = however. Additionally, it is often the case that in a casual setting, people will say O(...)when they mean Ɵ(...); this is technically true since the set of things Ɵ(exactlyThis) is a subset of O(noGreaterThanThis)... and it's easier to type. ;-)*

Big O notation is most commonly used by programmers as an approximate measure of how long a computation (algorithm) will take to complete expressed as a function of the size of the input set.

Big O is useful to compare how well two algorithms will scale up as the number of inputs is increased.

More precisely [Big O notation](http://en.wikipedia.org/wiki/Big_O_notation) is used to express the asymptotic behavior of a function. That means how the function behaves as it approaches infinity.

In many cases the "O" of an algorithm will fall into one of the following cases:

* **O(1)** - Time to complete is the same regardless of the size of input set. An example is accessing an array element by index.
* **O(Log N)** - Time to complete increases roughly in line with the log2(n). For example 1024 items takes roughly twice as long as 32 items, because Log2(1024) = 10 and Log2(32) = 5. An example is finding an item in a [binary search tree](http://en.wikipedia.org/wiki/Binary_search_tree) (BST).
* **O(N)** - Time to complete that scales linearly with the size of the input set. In other words if you double the number of items in the input set, the algorithm takes roughly twice as long. An example is counting the number of items in a linked list.
* **O(N Log N)** - Time to complete increases by the number of items times the result of Log2(N). An example of this is [heap sort](http://en.wikipedia.org/wiki/Heap_sort) and [quick sort](http://en.wikipedia.org/wiki/Quick_sort).
* **O(N^2)** - Time to complete is roughly equal to the square of the number of items. An example of this is [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort).
* **O(N!)** - Time to complete is the factorial of the input set. An example of this is the [traveling salesman problem brute-force solution](http://en.wikipedia.org/wiki/Travelling_salesman_problem).

Big O ignores factors that do not contribute in a meaningful way to the growth curve of a function as the input size increases towards infinity. This means that constants that are added to or multiplied by the function are simply ignored.

Big O is just a way to "Express" yourself in a common way, "How much time / space does it take to run my code?".

You may often see O(n), O(n^2), O(nlogn) and so forth, all these are just ways to show; How does an algorithm change?

O(n) means Big O is n, and now you might think, "What is n!?" Well "n" is the amount of elements. Imaging you want to search for an Item in an Array. You would have to look on Each element and as "Are you the correct element/item?" in the worst case, the item is at the last index, which means that it took as much time as there are items in the list, so to be generic, we say "oh hey, n is a fair given amount of values!".

So then you might understand what "n^2" means, but to be even more specific, play with the thought you have a simple, the simpliest of the sorting algorithms; bubblesort. This algorithm needs to look through the whole list, for each item.

My list

1

6

3

The flow here would be:

Compare 1 and 6, which is biggest? Ok 6 is in the right position, moving forward!

Compare 6 and 3, oh, 3 is less! Let's move that, Ok the list changed, we need to start from the begining now!

This is O n^2 because, you need to look at all items in the list there are "n" items. For each item, you look at all items once more, for comparing, this is also "n", so for every item, you look "n" times meaning n\*n = n^2

I hope this is as simple as you want it.

But remember, Big O is just a way to experss yourself in the manner of time and space.

**Big O describes the fundamental scaling nature of an algorithm.**

There is a lot of information that Big O does not tell you about a given algorithm. It cuts to the bone and gives only information about the scaling nature of an algorithm, specifically how the resource use (think time or memory) of an algorithm scales in response to the "input size".

Consider the difference between a steam engine and a rocket. They are not merely different varieties of the same thing (as, say, a Prius engine vs. a Lamborghini engine) but they are dramatically different kinds of propulsion systems, at their core. A steam engine may be faster than a toy rocket, but no steam piston engine will be able to achieve the speeds of an orbital launch vehicle. This is because these systems have different scaling characteristics with regards to the relation of fuel required ("resource usage") to reach a given speed ("input size").

Why is this so important? Because software deals with problems that may differ in size by factors up to a trillion. Consider that for a moment. The ratio between the speed necessary to travel to the Moon and human walking speed is less than 10,000:1, and that is absolutely tiny compared to the range in input sizes software may face. And because software may face an astronomical range in input sizes there is the potential for the Big O complexity of an algorithm, it's fundamental scaling nature, to trump any implementation details.

Consider the canonical sorting example. Bubble-sort is O(n^2) while merge-sort is O(n log n). Let's say you have two sorting applications, application A which uses bubble-sort and application B which uses merge-sort, and let's say that for input sizes of around 30 elements application A is 1,000x faster than application B at sorting. If you never have to sort much more than 30 elements then it's obvious that you should prefer application A, as it is much faster at these input sizes. However, if you find that you may have to sort ten million items then what you'd expect is that application B actually ends up being thousands of times faster than application A in this case, entirely due to the way each algorithm scales.

**A Plain English Explanation of the *Need* for Big-O Notation:**

When we program, we are trying to solve a problem. What we code is called an algorithm. Big O notation allows us to compare the worse case performance of our algorithms in a standardized way. Hardware specs vary over time and improvements in hardware can reduce the time it takes an algorithms to run. But replacing the hardware does not mean our algorithm is any better or improved over time, as our algorithm is still the same. So in order to allow us to compare different algorithms, to determine if one is better or not, we use Big O notation.

**A Plain English Explanation of *What* Big O Notation is:**

Not all algorithms run in the same amount of time, and can vary based on the number of items in the input, which we'll call *n*. Based on this, we consider the worse case analysis, or an upper-bound of the run-time as *n* get larger and larger. We must be aware of what *n* is, because many of the Big O notations reference it.

It is very difficult to measure the speed of software programs, and when we try, the answers can be very complex and filled with exceptions and special cases. This is a big problem, because all those exceptions and special cases are distracting and unhelpful when we want to compare two different programs with one another to find out which is "fastest".

As a result of all this unhelpful complexity, people try to describe the speed of software programs using the smallest and least complex (mathematical) expressions possible. These expressions are very very crude approximations: Although, with a bit of luck, they will capture the "essence" of whether a piece of software is fast or slow.

Because they are approximations, we use the letter "O" (Big Oh) in the expression, as a convention to signal to the reader that we are making a gross oversimplification. (And to make sure that nobody mistakenly thinks that the expression is in any way accurate).

If you read the "Oh" as meaning "on the order of" or "approximately" you will not go too far wrong. (I think the choice of the Big-Oh might have been an attempt at humour).

The only thing that these "Big-Oh" expressions try to do is to describe how much the software slows down as we increase the amount of data that the software has to process. If we double the amount of data that needs to be processed, does the software need twice as long to finish it's work? Ten times as long? In practice, there are a very limited number of big-Oh expressions that you will encounter and need to worry about:

The good:

* O(1) **Constant**: The program takes the same time to run no matter how big the input is.
* O(log n) **Logarithmic**: The program run-time increases only slowly, even with big increases in the size of the input.

The bad:

* O(n) **Linear**: The program run-time increases proportionally to the size of the input.
* O(n^k) **Polynomial**: - Processing time grows faster and faster - as a polynomial function - as the size of the input increases.

... and the ugly:

* O(k^n) **Exponential** The program run-time increases very quickly with even moderate increases in the size of the problem - it is only practical to process small data sets with exponential algorithms.
* O(n!) **Factorial** The program run-time will be longer than you can afford to wait for anything but the very smallest and most trivial-seeming datasets.

A Beginner’s Guide to Big O Notation

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the **worst-case** scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

Anyone who’s read Programming Pearls or any other Computer Science books and doesn’t have a grounding in Mathematics will have hit a wall when they reached chapters that mention O(N log N) or other seemingly crazy syntax. Hopefully this article will help you gain an understanding of the basics of Big O and Logarithms.

As a programmer first and a mathematician second (or maybe third or fourth) I found the best way to understand Big O thoroughly was to produce some examples in code. So, below are some common orders of growth along with descriptions and examples where possible.

O(1)

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

bool IsFirstElementNull(String[] strings)

{

if(strings[0] == null)

{

return true;

}

return false;

}

O(N)

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

bool ContainsValue(String[] strings, String value)

{

for(int i = 0; i < strings.Length; i++)

{

if(strings[i] == value)

{

return true;

}

}

return false;

}

O(N2)

O(N2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N3), O(N4) etc.

bool ContainsDuplicates(String[] strings)

{

for(int i = 0; i < strings.Length; i++)

{

for(int j = 0; j < strings.Length; j++)

{

if(i == j) // Don't compare with self

{

continue;

}

if(strings[i] == strings[j])

{

return true;

}

}

}

return false;

}

O(2N)

O(2N) denotes an algorithm whose growth will double with each additional element in the input data set. The execution time of an O(2N) function will quickly become very large.

Logarithms

Logarithms are slightly trickier to explain so I’ll use a common example:

[Binary search](http://en.wikipedia.org/wiki/Binary_search) is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower than the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set.

This type of algorithm is described as **O(log N)**. The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds. Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets.

**What is order notation (or Big "O" notation)?**

    Order notation, or Big "O" notation, is a measure of the running time of an algorithm, as it relates to the size of the input to that algorithm.  It is intended, not to measure the performance of the machine on which the algorithm is run, but rather to strictly measure the performance of the algorithm itself.   Thus, since different machines can vary in their speeds by some constant factor, we remove all constant factors from consideration when we talk about order notation.  For example O(2) and O(1) are considered to be the same.  Similarly, O(n) is the same as O(2n), and the same as O(100n)

**Some polynomial running times**

**O(1)**   
    An algorithm with this running time is said to have "constant" running time.  Basically, this means the algorithm always take about the same amount of time, regardless of the size of the input.  To state it technically, if an algorithm will never perform more than a certain number of steps, no matter how large the input gets, then that algorithm is considered to have a constant running time.  For example, an algorithm which consists of performing exactly 7 multiplication's has a constant running time.  An algorithm which always finishes in under a year has a constant running time.  Although constant time is the best running time an algorithm can have, that algorithm could still be considered bad if the total amount of time to run the algorithm were too large, perhaps because there were many complex or unnecessary steps in the algorithm.   
    Some examples of O(1) algorithms include: inserting an element onto the front of a linked list, popping from or pushing onto a stack, and retrieving the nth element of an array.

**O(n)**   
    An algorithm which runs in O(n) is said to have a "linear" running time.  This basically means that the amount of time to run the algorithm is proportional to the size of the input.  To be technical, an algorithm which never performs more than certain number of steps for each element in the input has a linear running time.  For example, an algorithm which sums the total of a list of numbers has a linear running time, because the number of additions required is the same as the number of elements (thus there is 1 addition for every element).   
    Some examples of O(n) algorithms include searching through an unordered list, incrementing every element of an array, and calculating fibonacci numbers using dynamic programming.  There is also a clever way to find the median element of a list in linear time.

**O(n2)**   
    An algorithm with this running time is said to have "quadratic" running time.  This means that whenever you increase the size of the input by a factor of n, the running time increases by a factor of n2.  For example, if you double the size of the input of a quadratic algorithm, then the running time will quadruple.   
  Some sorting algorithms, such as insertion sort and bubble sort, have quadratic running times.

**O(lgn)**   
    An algorithm with O(lgn) running time is said to have "logarithmic" running time.  This means that as the size of the input increases by a factor of n, the running time increases by a factor of the logarithm of n.  For example, if you increase the input size of a O(lgn) algorithm by a factor of 1024, the running time will increase by a factor of 10.  This running time is better than O(n), but not as good as O(1).  As the input size gets large, however, the behavior becomes comparable to O(1) in many circumstances.   
    Algorithms which search through ordered lists or binary trees, as well as operations on heaps generally have logarithmic running times.

**O(nlgn)**   
    An algorithm which has this order, will in increase in running time proportionate to the size of the input times the logarithm of the size of the input.  Technically speaking, an algorithm which when given an input of size n never performs more than cnlgn steps (for some c which is always the same regardless of the value of n) has a running time of O(nlgn).  This running time is better than O(n2) but not quite as good as O(n).   
    The fastest sorting algorithms, including mergesort and quicksort, have O(nlgn) running times

**Worse than polynomial running times**

**O(2n)**   
    An algorithm with this running time is said to be "exponential".  This means that its running time will double every time you add another element to the input.  An algorithm with this running time is generally considered to be too slow to be useful for anything but the smallest of problems.  For example, an O(2n) algorithm which takes an input with 30 elements may need to perform as many as 1 billion steps.  If the input has 40 elements then the 1 trillion steps may be necessary.  No computer in the world can do this in a reasonable amount of time.

**O(n!)**   
    An algorithm with this running time is said to be "factorial".  This is worse than exponential.  This means that if the algorithms take an input of size n, the total time will be proportional to n\*(n-1)\*(n-2)\*...\*2\*1.  For example, if an algorithm with this running time were to take 8 elements in its input, the number of steps would be proportional to 8\*7\*6\*5\*4\*3\*2\*1 = 40320.  When the input size reaches 15, the number of steps may exceed 1 trillion.  An example of a factorial algorithm is one that calculates fibonacci numbers recursively.

**O(nn)**   
    This running time is even worse than factorial.  An algorithm with this running time which takes 10 elements of input may need to perform 10 billion steps.

**Almost constant running times**

**O(lg\*n)**   
    This running time is called "log-star" time.  The log-star function calculates how many times you would need to take the log of n before you would go below 2.  For example: lg\*4 = 2,  lg\*16=3, lg\*65536=4. lg\* 1000000000000000 < 5.  This function grows so slowly, that for all practical purposes it may be considered constant.  Technically it is not constant, but no computer in the world can store enough data to cause the total running time to increase more than a factor of 5 of the total running time which the algorithm takes when the input has just 2 elements.

**O((m,n))**   
    This function, which is called the inverse of Ackerman's function, performs similarly to the log-star function.  If m=2 then this is equivalent to the log-star function, and if m>2 then this grows even more slowly.   
  