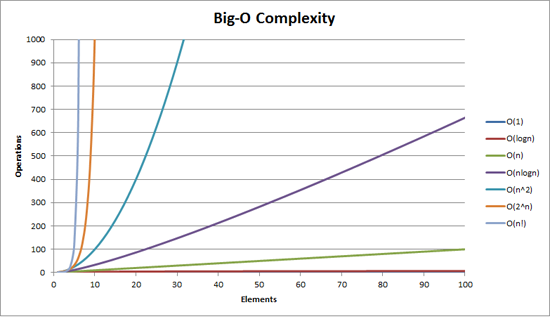
# Big-O Complexity

Quick note, 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 a two-side bound). In my experience this is actually typical of discussions in non-academic settings. Apologies for any confusion caused.

Big O complexity can be visualized with this graph:



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 given the phone number (reverse lookup):

* **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**.

O(n), O(n2) etc are all polynomial time. Some problems cannot be solved in polynomial time. 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).

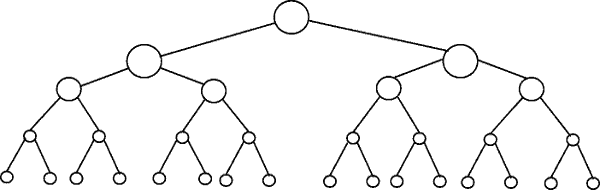
# [What does O(log n) mean exactly?](https://stackoverflow.com/questions/2307283/what-does-olog-n-mean-exactly)

what exactly is O(log n)? For example, what does it mean to say that the height of a complete binary tree is O(log n)?

I do know (maybe not in great detail) what Logarithm is, in the sense that: log10 100 = 2, but I cannot understand how to identify a function with a logarithmic time.

What does it mean to say that the height of a complete binary tree is O(log n)?

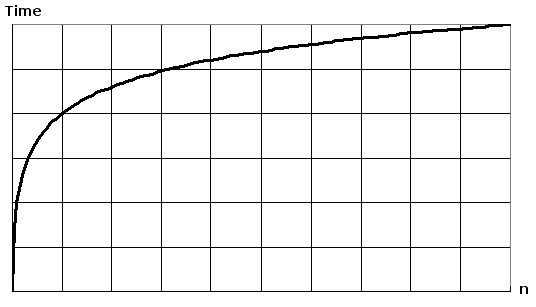
The following drawing depicts a binary tree. Notice how each level contains double the number of nodes compared to the level above (hence binary):

[](https://i.stack.imgur.com/ZsiDW.png)

Binary search is an example with complexity O(log n). Let's say that the nodes in the bottom level of the tree in figure 1 represents items in some sorted collection. Binary search is a divide-and-conquer algorithm, and the drawing shows how we will need (at most) 4 comparisons to find the record we are searching for in this 16 item dataset.

Assume we had instead a dataset with 32 elements. Continue the drawing above to find that we will now need 5 comparisons to find what we are searching for, as the tree has only grown one level deeper when we multiplied the amount of data. As a result, the complexity of the algorithm can be described as a logarithmic order.

Plotting log(n) on a plain piece of paper, will result in a graph where the rise of the curve decelerates as n increases:

[](https://i.stack.imgur.com/qPNNp.png)

Another Explanation:

|  |  |
| --- | --- |
|  | I cannot understand how to identify a function with a log time.  The most common attributes of logarithmic running-time function are that:   * the choice of the next element on which to perform some action is one of several possibilities, and * only one will need to be chosen.   or   * the elements on which the action is performed are digits of n   This is why, for example, looking up people in a phone book is O(log n). You don't need to check *every* person in the phone book to find the right one; instead, you can simply divide-and-conquer by looking based on where there name is alphabetically, and in every section you only need to explore a subset of the each section before you eventually find someone's phone number.  Of course, a bigger phone book will still take you a longer time, but it won't grow as quickly as the proportional increase in the additional size.  We can expand the phone book example to compare other kinds of operations and *their* running time. We will assume our phone book has *businesses* (the "Yellow Pages") which have unique names and *people* (the "White Pages") which may not have unique names. A phone number is assigned to at most one person or business. We will also assume that it takes constant time to flip to a specific page.  Here are the running times of some operations we might perform on the phone book, from best to worst:   * **O(1) (worst case) (**Means Worst Case is as Good as Best Case**):** Given the page that a business's name is on and the business name, find the phone number. * **O(1) (average case):** Given the page that a person's name is on and their name, find the phone number. * **O(log n):** Given a person's name, find the phone number by picking a random point about halfway through the part of the book you haven't searched yet, then checking to see whether the person's name is at that point. Then repeat the process about halfway through the part of the book where the person's name lies. (This is a binary search for a person's name.) * **O(n):** Find all people whose phone numbers contain the digit "5". * **O(n):** Given a phone number, find the person or business with that number. * **O(n log n):** There was a mix-up at the printer's office, and our phone book had all its pages inserted in a random order. Fix the ordering so that it's correct by looking at the first name on each page and then putting that page in the appropriate spot in a new, empty phone book.   For the below examples, we're now at the printer's office. Phone books are waiting to be mailed to each resident or business, and there's a sticker on each phone book identifying where it should be mailed to. Every person or business gets one phone book.   * **O(n log n):** We want to personalize the phone book, so we're going to find each person or business's name in their designated copy, then circle their name in the book and write a short thank-you note for their patronage.   Finding a person in book is O(logn), doing it for n people is O(nlogn)   * **O(n2):** A mistake occurred at the office, and every entry in each of the phone books has an extra "0" at the end of the phone number. Take some white-out and remove each zero.   Check every entry in 1 phone book is O(n), doing it for n books is O(n\*n) = O(n2)   * **O(n · n!):** We're ready to load the phonebooks onto the shipping dock. Unfortunately, the robot that was supposed to load the books has gone haywire: it's putting the books onto the truck in a random order! Even worse, it loads all the books onto the truck, then checks to see if they're in the right order, and if not, it unloads them and starts over. (This is the dreaded [**bogo sort**](http://en.wikipedia.org/wiki/Bogosort).)   Permutation of arranging n books in order is O(n!), doing it n times (as robot is unloading everything and starting fresh) is O(n.n!)   * **O(nn):** You fix the robot so that it's loading things correctly. The next day, one of your co-workers plays a prank on you and wires the loading dock robot to the automated printing systems. Every time the robot goes to load an original book, the factory printer makes a duplicate run of all the phonebooks! Fortunately, the robot's bug-detection systems are sophisticated enough that the robot doesn't try printing even more copies when it encounters a duplicate book for loading, but it still has to load every original and duplicate book that's been printed. |