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(IList<string> elements)
{
    return elements[0] == null;
}
```

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(IList<string> elements, string value)
{
    foreach (var element in elements)
    {
        if (element == value) return true;
    }

    return false;
}
```

$O(N^2)$

$O(N^2)$ 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 $O(N^3)$, $O(N^4)$ etc.

```
bool ContainsDuplicates(IList<string> elements)
{
```

```
    for (var outer = 0; outer < elements.Count; outer++)
    {
        for (var inner = 0; inner < elements.Count; inner++)
        {
            // Don't compare with self
            if (outer == inner) continue;

            if (elements[outer] == elements[inner]) return true;
        }
    }

    return false;
}
```

$O(2^N)$

$O(2^N)$ denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an $O(2^N)$ function is exponential - starting off very shallow, then rising meteorically. An example of an $O(2^N)$ function is the recursive calculation of Fibonacci numbers:

```
int Fibonacci(int number)
{
    if (number <= 1) return number;

    return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

Logarithms

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

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.

This article only covers the very basics or Big O and logarithms. For a more in-depth explanation take a look at their respective Wikipedia entries: Big O Notation, Logarithms.

Published 23 June 2009
Category Articles
Tagged Computer Science, Programming

26 comments

Horacio says:

Excellent explanation, big O notation is sometimes hard to “imagine” when teaching but this examples couldn’t have made it easier

Jeff Aigner says:

It might be worth mentioning that O(log N) is the theoretical limit for searching a data set.

Matt S says:

I remember learning Big-O in college and it went right over my head. Your article gave me a better understanding than any of my professor’s ever did. Thanks!

bluehavana says:

O(log N) is log base 2 as well. This is an important distinction.

probabilityzero says:

When my programming professor went over this during our intro to programming class, that’s about all she said about the subject. Seeing O(logN) looks scary if you don’t know what it means, but it isn’t really that difficult a concept to pick up.

Mylo says:

A very helpful and easy to understand explanation of the Big O - thanks for providing this. I always find code examples make understanding the problem easier!

Al K. says:

Nice article, but I'll have to quibble with this: "Big O specifically describes the worst-case scenario". That is not the case. Big-O notation just describes asymptotic bounds, so it is correct to say something like, for example, "Quicksort is in O(n!)," even though Quicksort's actual worst-case running time will never exceed O(n^2). All Big-O is saying is "for an input of size n, there

is a value of n after which quicksort will always take less than n! steps to complete." It does not say "Quicksort will take n! steps in the worst case."

Dan says:

Thanks for writing this article and using programming code as an example. It was very insightful and useful!

Priyanka Tyagi says:

Very nice article!

Evelyn says:

A graph showing each of these at the same time would be helpful. This one is in the right region, but bad: http://therecyclebin.files.wordpress.com/2008/05/time-complexity.png (why the different scales?). This one is better, but doesn't have all of your examples: http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/index.html

Gregory Kornblum says:

Very well put. It's a shame people concern themselves more with design patterns and domain models while not giving a bit of attention to the core of all programming and where most of its concepts and methods are born, mathematical logic. Of course they all play an equal role in development but none should never be forgotten as often as the mathematical logic is.

hiddenson says:

Thank you for your crystal clear guide. I studied some of this concepts so long ago that it almost seems another life :) PS: I knew about the binary search under another name: the dichotomic search.

Steve says:

Nice post. I like the code examples to add context to what you're describing.

Josh W. says:

It might be worth mentioning that O(log N) is the theoretical limit for searching a data set.

Michael Clark says:

@bluehavana "O(log N) is log base 2 as well. This is an important distinction." Regarding O notation there is no distinction between O(log N), O(ln N) or logarithms of any base. This is because O notation is saying, asymptotically, that the algorithm has a time complexity limited by the log function. A logarithm of any base b is a real multiple of any generic logarithm: log_b N = log N / log b. These are equivalent in O notation because of its definition: "if g(x) is O(f(x)) there exists some real constant M such that g(x) <= M f(x)" obviously if we multiply by 1/log b, there is another real constant N = M / log b that satisfies this definition. Anyway nice article Rob, hopefully I'll be joining you for a few lunches up in London soon.

arnuld says:

Short and accurate content. Its was so much easier with source code and not to mention the quite good explanation of O(log N). Thanks for this :)

kai says:

Great article! This made understanding the Big O notation so much easier. I'm looking forward to reading more blog posts.

Saman says:

Very nice article for beginners who want to gain understanding of Big Oh. Good work.

Mick says:

Fantastic - you want to come and be our lecturer. As a mature computing student with no grounding in mathematics I was really hitting the wall with this.. The O and big O I can grip but finding n within a program was the problem. The examples are great and I now have a better angle on this... Thanks

Georgi says:

I found this to be an excellent introductory post and have used it as an inspiration for my own post where ive included some examples in java. Thanks a lot :) http://nerdgerl.wordpress.com/2009/11/18/an-intro-to-big-oh-notation-with-java/

ktj says:

This is a brilliant piece of work, very easy to understand. Most programming lecturers/professors at university have serious hardships in trying to explain the big O notation.

Joseph says:

Thank you. Nice article. I've tired of reading textbooks with cryptic writing and lack of introduction to Big O and logarithms. I'm still on the struggle, but your posts has given me a foundation in which I'm very grateful.

Karim Naufal says:

Hey, great explanation man! Simple and straightforward... Even MIT teachers in there video course (check out http://academicearth.org) couldn't explain it so simply. Thanks a lot and keep up the good work. Cheers!

Anjaneai says:

This was a great lesson. Good Example. usually people get stuck understanding and relating Mathematical Notations Functions and Complex Equations with Code. You did a great job building a bridge in between! Kudos! Thanks!

Andrew says:

You made this particularly complicated subject very simple. I was taught this in a college course, but it didn't make any sense. I finally get it after reading your article, thank you for writing this.

Puneet says:

Excellent. You beauty. You made my day. Being an average in Maths, it was increasingly difficult to get this big O notation. Thanks, I am bookmarking. Don't get rid of this page, ever.

About

Rob Bell is a software developer from London specialising in web development with an interest in most things nerdy.

More

GitHub LinkedIn Contact me