What is the big O notation and how do I calculate it?

A lot of people I see seem to know off the top of their head whether a process is O(log n) or O(1) and so on. But I don't even know the slightest on how to determine that. How do I calculate the big O notation? Can you give examples of code and their corresponding representations in O notation?

142 FOLLOWERS

Last asked: 26 Jul

QUESTION TOPICS

- Big-O Notation
- Computational Complexity Theory
- Data Structures
- Algorithms
- Computer Science

QUESTION STATS

Views	78,836
Followers	142
Merged Questions	2
Edits	

RELATED QUESTIONS

- What is Big O notation?
- I am not able to understand anything about algorithms like big o, big omega and big teta notation. What should I do?
- Is big-O notation taught in discrete mathematics?
- What does it mean when a function is followed by big-O notation?
- How can I explain Big O notation to a non-mathematician?
- Computer Programming: What is the big O notation?
- How do I calculate the Big-oh notation of a function?
- Big-O Notation: What are examples of good benchmarking tools to determine the growth in time and calculations needed for a...
- How do I find the big-O for the following function?
- What would be the Big Omega notation of insertion sort?

11 Answers

Gayle Laakmann McDowell, Author of Cracking the Coding Interview
70.2k Views • Upvoted by Vladimir Novakovski, IOI 2001, ACM World Finals 2003 and 2004
Gayle is a Most Viewed Writer in Data Structures.

Big-O notation is a way to express the efficiency of an algorithm. If you're going to be working with code, it is extremely important that you understand Big-O. It is, quite literally, the language we use to express efficiency.

Big-O is an expression of how the execution time of a program scales with the input data. That is, as the input gets bigger, how much longer will the program take? Just a little bit? A lot longer?

Suppose you have a function *foo* which does some processing on an array of size *N*. If *foo* takes O(N) time, then, as the array grows (that is, as *N* increases), the number of seconds *foo* takes will also increase in some sort of linear fashion.

NOTE: *This is an excerpt (or will be, once it's published) from Cracking the PM Interview. If you find errors (even little typos -- I haven't proofread this yet), or anything that is a bit confusing or misleading, please comment / correct.*

Interested in the book, Cracking the PM Interview? Sign up on the website to be notified when it launches.

Real-Life Big-O

Many "operations" in real life are O(N). Driving, for example, can be thought of as O(N). As the distance *N* increases, driving time also increases in a linear fashion.

What might not be O(N)?

Imagine we invited a bunch of people (including you) to a dinner party. If I invited twice as many people to the party, you will have to shake twice as many hands. The time it will take *you* to shake everyone's hand can be expressed as O(N). If I double the amount of guests, it will take you twice as long. This is a linear, or O(N), increase.

Now, let's suppose everyone wants to shake hands, but for some strange reason only one pair can shake hands at a time. As N increases, how much longer will this meet and greet take? Well, your work will take O(N) time -- but so will everyone else's. The time it takes increases *proportionally* with O(N^2), since there are roughly N^2 pairs.

Dropping Constants

If you are paying close attention, you might say, "But wait! There aren't N^2 pairs. People aren't shaking hands with themselves, and you're double counting every pair. There are really N(N-1)/2 pairs. So we should say O(N(N-1)/2).”

You’re absolutely right. There *are* N(N-1)/2 pairs (which is .5*N^2 - .5N), but we still say that this is O(N^2).

Big-O is very hand-wavey, wishy-washy. We’re trying to express how the time changes in rough terms, not a offer a precise calculation for the number of seconds something takes.

As a result, we drop constant factors, so O(2N) is the same as O(N). We also drop the addition or subtraction of constants, so O(N - 5) becomes O(N). Put together, these two factors mean that O(N^2 + N) should be written as O(N^2). Think about it: if O(N^2) and O(N^2 + N^2) are the same, then O(N^2 + N), which is between those two, should be treated as the same.

This is a very important thing to understand. You should never express an algorithm as “O(2N).” This is not a “more precise” or “better” answer than O(N); it’s only a confusing one. A so-called O(2N) algorithm is O(N) and should be expressed as such.

Which of the below expressions are equivalent to O(N^3)?

- O(3N^3)
- O(N(N^2 + 3))
- O(N^3 - 2)
- O(N^3 + N lg N)
- O(N^3 - N^2 + N)
- O((N^2 + 3)(N+1))

All of them!

Drop your constants and just keep the most important term.

Multiple Variables

Back to the handshaking example. Suppose we invited men and women to our dinner party. The men all already know each other and the women all already know each other. Therefore, people will only shake hands with the opposite gender.

Assuming that we’re still in bizarro land where only one pair can shake hands at a time, how would you express how long this takes?

Don’t say O(N^2). Suppose we have 100 men and 1 woman. Adding one man will add one handshake, but adding one woman will add 100 handshakes. The time it takes does not actually increase proportional to the number of people squared.

These are different “variables” and it matters which one we increase. The correct way to express this is with two variables. If there are *M* men and *W* women, then our meet and greet takes O(M*W) time.

What if the women all knew each other, but the men knew no one at all? We would then say that the meet and greet is O(M^2 + M*W). Note that we do not drop that extra M*W term; it’s a different variable, and it matters.

Why This Matters (And Why It Doesn’t)

Let’s suppose that we have two functions which process some data. The function *foo* takes O(N) time and the function *bar* takes O(N^2) time. On a given data set, which one will be faster?

We don’t know, actually.

The runtime of *foo* will increase proportionally to O(N) and the runtime of bar will increase proportionally to O(N^2). So, eventually, the O(N^2) line should exceed the O(N) time.

However, we can't make any determinations on a particular data set. The O(N^2) could be faster on smaller data sets; it might not have yet exceeded the O(N) line. Plus, even on very large data -- after this "overtaking" occurs -- there could be exceptions. Maybe, when N is divisible by 1000, the *bar* code will hit a special case and suddenly operate very quickly. We just don't know.

This doesn't make Big-O useless; we just have to be very careful about how we apply it.

Big-O allows us to say things like, "In general, as our data set grows in size, this algorithm will be much, much faster than this other one." It also allows us to say, "You want to run this O(N^2) algorithm, and N is the number of files on our network? Sorry, that's just not going to work." That matters -- a lot.

Moreover, it gives us a language for expressing efficiency that's isn't reliant on the system architecture or the technologies used. Without big-O, we'd likely have to discuss efficiency in terms of seconds, which has little meaning when you are on a different system.

problems as O(log(N)) or O(lg(N)) but we aren't particularly concerned about specifying whether we mean log-base-2 or log-base-10. That's because it doesn't matter. The difference between one log and another is just a constant factor. Since big-O time doesn't care about constant factors, we don't need to care about what our log base is.

Carrier Pigeons and Data Transfer

In 2009, a South African company named The Unlimited grew frustrated by their ISP's slow internet and made news by comically showing just how bad it is. They "raced" a carrier pigeon against their ISP. The pigeon had a USB stick affixed to its leg and was taught to fly to an office, 50 miles away. Meanwhile, the company transferred this same data over the internet to this same office. The pigeon won -- by a long shot.

If you've been paying attention, you'll see the problem here. With a sufficiently large data set, the pigeon will always win; pigeon data transfer is O(1) and internet data transfer is O(N). Likewise, with a sufficiently small data set, the pigeon would have lost. The difference between fast internet and slow internet is just where those lines cross.

Big-O Space and More

The concept of big-O can be used for much more than runtime. In fact, very commonly it is used to describe how much memory an algorithm uses.

For example, suppose I have an algorithm that creates and initializes an *NxN* matrix:

```
int[][] a = new int[N][N]; /* NxN matrix
for i from 0 to N {
   for j from 0 to N {
      a[i][j] = i + j
   }
}
```

This algorithm takes O(N^2) time and O(N^2) space.

Sample Problems

Now, let's move on to some examples (in pseudocode). Can you find the runtime of each of these problems?

Example 1
Consider the following code to print the numbers from 0 to n.

```
for i from 0 to n {
   print i /* line 2 */
}
```

This is said to be O(n) time. That is, if we were to run this code for many different values of n, the runtime would increase at a rate proportional to n.

Example 2
What about this code?

```
1 sum = 0
2 for i from 0 to n {
3    sum = sum + i
4    for j from 0 to n {
5       sum = sum + j /* line 5 */
6    }
7 }
```

This is O(N^2) time. There are two for-loops, each running from 0 to n. How many times does line 5 get executed? O(N^2). The time for this code to run will increase at a rate of O(N^2).

Example 3
The code below uses two variables. What is its running time?

```
/* Assume A and B are both arrays.*/
for i from 0 to A.length {
   int j = 0;
   while (a[i] != b[j]) {
      print a[i]
      j = j + 1
   }
}
```

This is said to be O(a*b), where a is the length of A and b is the length of B. Although the inner while loop may sometimes terminate early (having found *a[i]*), the expected case is that it will iterate through roughly all of *B*.

Example 4
Here is a more challenging example.

```
int i = N;
while i >= 1 {
   print i
   i = i / 2
}
```

We need to think about what this for loop will do. This for loop will do something (print a value) and then continuously divide by 2 until it gets below 1.

How many times can we divide by *N* by 2 until we get below 1? If we approached this in reverse, we could say: how many times can we multiply 1 by 2 until we get to N? This would be the value *x*, where 2^x = n. This for loop, therefore, iterates *x* times.

Now we just need to solve for x:
  2^x = n
  log(2^x) = log(n)
  x log(2) = log(n)
  x = log(n) / log(2)
So, this code operates in O(log(n)) time.

Note: If you've taken an algorithms class, you might remember that, technically, big-O refers to an upper-bound. Anything that is O(N) could also be said to be O(N^2). To describe the exact runtime, we should be using big-theta.

That is true, by the official mathematical definition of big-O. However, outside of an algorithms class, this distinction has been forgotten about.

Updated 17 Sep 2013 • View Upvotes

More Answers Below. **Related Questions**

What is Big O notation?

I am not able to understand anything about algorithms like big o, big omega and big teta notation. What should I do?

Is big-O notation taught in discrete mathematics?

What does it mean when a function is followed by big-O notation?

How can I explain Big O notation to a non-mathematician?

Kk Aravind Bharathy, Web developer, Writer
7.5k Views • Kk is a Most Viewed Writer in Computational Complexity Theory.

Asymptotic analysis is used to study how the running time grows as size of input increases.This growth is studied in terms of the input size.Input size, which is usually denoted as N or M, it could mean anything from number of numbers(as in sorting), number of nodes(as in graphs) or even number of bits(as in multiplication of two numbers).

While dealing with asymptotic analysis our goal is find out which algorithm fares better in specific cases.Realize that an algorithm runs on quite varying times even for same sized inputs.To appreciate this, consider you are a sorting machine.You will be given a set of numbers and you need to sort them.If I give you a sorted list of numbers, you would have no work, and you are done already.If I gave you a reverse sorted list of numbers, imagine the number of operations you need to do to make the list sorted.Now that you see this, realize that we need a way of knowing what case the input would be?Would it be a best case?Would I get a worst case input?To answer this, we need some knowledge of the distribution of the input.Will it all be worst cases?Or would it be average cases?Or would it mostly be best cases?

The knowledge of the input distribution is fairly difficult to ascertain in most cases.Then we are left with two options.Either we can assume average case all the time and analyze our algorithm, or we could get a guarantee on the running case irrespective of the input distribution.The former is referred to as average case analysis, and to do such an analysis would require a formal definition of what makes an average case.Sometimes this is difficult to define and requires much mathematical insight.All the trouble is worth it, when you know that some algorithm runs much faster on the average case, compared to its worst case running time.There are several randomized algorithms that stand testimony to this.in such cases, doing an average case analysis reveals its practical applicability. The latter, the worst case analysis is more often used since it provides a nice guarantee on the running time.In practice coming up with the worst case scenario is often fairly intuitive.Say you are the sorting machine, worst case is like reverse sorted array.What's the average case?
Yup, you are thinking, right?Not so intuitive.

The best case analysis is rarely used as one does not always get best cases.Still one can do such an analysis and find interesting behavior.

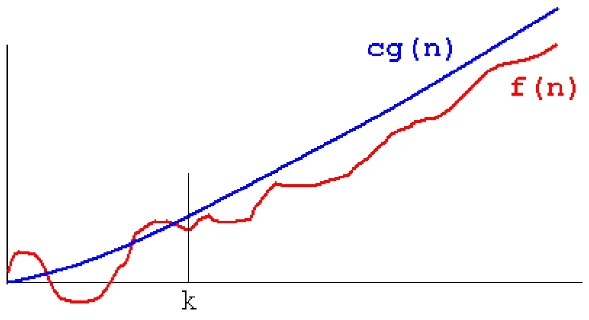
In conclusion, when we have a problem that we wanna solve, we come up with algorithms.Once we have an algorithm, we need to decide if it's of any practical use to our situation.If so we go ahead and shortlist the algorithms that can be applied, and compare them based on their time and space complexity.There could be more metrics for comparison, but these two are fundamental.One such metric could be ease of implementation.And depending on the situation at hand you would employ either worst case analysis or average case analysis or best case analysis.For example if you rarely have worst case scenarios, then its makes much more sense to carry out average case analysis.However if the performance of our code is of critical nature and we need to provide the output in a strict time limit, then its much more prudent to look at worst case analysis.Thus, the analysis that you make depends on the situation at hand, and with time, the intuition of which analysis to apply becomes second nature.

Big-OH(in context of this SO question:http://stackoverflow.com/questio...

When we say f(x)=O(g(x)), we mean that for some constant c independent of input size,

f(x)<=c.g(x) for all x>=k

in other words, beyond a certain point k, the curve f(n) is always bounded above by the curve g(n) as shown in the figure.



Now in the case you have considered, the operations of addition and subtraction, multiplication are all primitive operations that take constant time(O(1)). Let's say the addition of two numbers takes 'a' time and assigning the result takes 'b' time.

So for this code:

```
 for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    a[i,j]=b[i,j]+c[i,j]
```

Let's be sloppy and ignore the for loop operations of assignment and update. The running time T(n)=(a+b)n2.

Notice that this is just O(n2), why?

As per the definition, this means we can identify some point k beyond which for some constant c the curve T(n) is always bounded above by n2.

Realize that this is indeed true.We can always pick sufficiently large constants so that c.n^2 curve always bounds above the given curve.

This is why people say:Drop the constants!

Bottomline: Big O of f(n) is g(n) means, to the right of some vertical line, the curve f(n) is always bounded above by g(n).

A few more posts to help you:
Page on Stackoverflow
Page on Stackoverflow

Updated 28 Aug 2014 • View Upvotes

Ezequiel H. Martinez, Computer Scientist & MBA with 20 years of experience.
2.1k Views

Definition

> Big O function, is the expected growth function of an algorithm for the amount of data (to be processed) in its worst case scenario.

N, is always the amount of data to be processed. So, an algorithm of time O(N), will do as you have to check every element N times. Linear.
O(N^2), will be like you've to check every N element, N times!!! HUGE.

So, if you have the following code (javascript):

```
var i=0;
var n=10;

for (i=1;i<n;i++){
    console.log(i);
}
```

Is O(n), cuz you iterate along the data n times.

Something that is O(n^2) (N squared) would be an iteration inside an iteration.
Like:

```
var i=0;
var x=0;
var n=10;

for (i=1;i<n;i++){
    for (x=1;x<n;x++){
        console.log(i);
    }
}
```

Every additional for/while/do, inside another inner one, will add an extra power (like, N^2, N^3, N^4, etc...).

If, in the other hand, you made one after the other:

```
var i=0;
var x=0;
var n=10;

for (i=1;i<n;i++){
    console.log(i);
}
for (x=1;x<n;x++){
     console.log(i);
}
```

It N+N, equals? 2N. Since N is the higher order of magnitude, we say this algorithm is a "O(n)".

And what about the following?

```
var i=0;
var x=0;
var n=10;
var m=10;

for (i=1;i<n;i++){
    console.log(i);
}
for (x=1;x<m;x++){
     console.log(i);
}
```

Well, that should be O(n+m), right? no, is O(n). Because the growth is linear.

See? is the function GROWTH what matters.

You have, actually, three kinds of algorithms, and not much more:

1. O(n): Linear (all functions of the like: f(x)= kX + C, and the like)
2. O(N * log N): logarithmic grow (anything that entails trees usually)
3. O(n^x): exponential. (Or a polinomial, like: kn^x + ... nm + i; too costly, avoid this at all cost)

Usually, intuitively we will fall in n or n^x cases. Computer Scientist where struggling for more than 60 years about how to convert an exponential problem in a linear one.

The best thing you can do, is to bring an algorithm that transforms it from N^x in a logarithmic one. Usually, this sort of solution entails the usage of trees, ordered lists (and search algorithms of the kind "divide and conquer" like quicksort) or other special data structures.

That's what Computer Science is for. :-)

Written 26 Aug • View Upvotes

Quora User, Hobbyist Programmer
7.5k Views

Let's say you have two functions on the real line - f and g so that takes f takes a real number and produces another one. Similar for g.

Then f is said to be O(g) if there is a x_0 and constant c such that for all x > x_0, f(x) <= c * g(x)

So for all values of x greater than x_0, f(x) is bounded by a constant times g(x).

For example, if you consider a polynomial f(x) = x^2+x+1. Then clearly for all x>1, x+1 < x^2

which means f(x) <= x^2+x^2 and therefore f(x) <= 2x^2 for all x > 1. Therefore, the polynomial f(x) is O(x^2).

Now O notation is not limited to functions on the real line, but I have used it here for simple examples.

A great example of a simple algorithm with its O properties is here - Insertion sort

Written 12 Nov 2012 • View Upvotes

Daniel R. Page, Theoretical Computer Scientist
2.3k Views • Daniel is a Most Viewed Writer in Big-O Notation.

You calculate it as follows.

1) Identify a barometer in your pseudocode that is an elementary operation (e.g., arithmetic assignment). No line in your code will occur more times than this one with respect to the input.
2) Count the number of times that line occurs. You'll get a function with respect to the input size.
3) Then you must determine what it's asymptotics are. There are many ways to do that, here are two that are normally covered in an undergraduate course on analysis of algorithms.

a) Apply the Limit Theorem for Big-oh.
b) Prove directly using the definition of Big-oh to determine membership.

Following this, you want to refine your asymptotic result using properties that follow from the definition of Big-oh. For example, if there are constants e.g., 2 in your growth function, you can drop these as the input size gets large, they become irrelevant.

Written 8 Mar • View Upvotes

Related Questions

Computer Programming: What is the big O notation?

How do I calculate the Big-oh notation of a function?

Big-O Notation: What are examples of good benchmarking tools to determine the growth in time and calculations needed for an algorithm?

How do I find the big-O for the following function?

What would be the Big Omega notation of insertion sort?

What's the best way to explain big-O notation in laymen's terms?

What is the solution for big O notation for function?

What are some algorithms that are famous for their big O notations?

Of all the notations, why do we use big o to calculate time complexity of a program?

Reviews of: Big-O Notation

What does n_0 mean when describing Big-O notation?

What is the concept of the big O notation? How is it useful in the analysis of algorithms?

How do I prove or disprove the following statements about f(n), g(n), and h(n) in the context of Big-Oh notation?

What are the basic differences between Big O and Big Omega notations?

What is the way to solve questions of Big O Notation in analysis of algorithms?

Top Stories