What is Big O notation? Do you use it? [duplicate]

Asked 16 years, 3 months ago Modified 11 years, 3 months ago

Viewed 34k times

36

This question already has answers here:

What is a plain English explanation of "Big O" notation? (43 answers)

Closed 11 years ago.

What is Big O notation? Do you use it?

I missed this university class I guess :D

Does anyone use it and give some real life examples of where they used it?

See also:

Big-O for Eight Year Olds?
Big O, how do you calculate/approximate it?
Did you apply computational complexity theory in real life?

optimization complexity-theory big-o

Share

Improve this question

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 • 1

asked Sep 25, 2008 at 12:29

Brian G

54.9k • 58 • 127 • 141

8 Sure, whenever I'm talking about my love life – Danimal Sep 25, 2008 at 12:33

12 Answers

Sorted by: Highest score (default)

50

One important thing most people forget when talking about Big-O, thus I feel the need to mention that:

You cannot use Big-O to compare the **speed** of two algorithms. Big-O only says how much slower an algorithm will get (approximately) if you double the number of items processed, or how much faster it will get if you cut the number in half.

However, if you have two entirely different algorithms and one (`A`) is `O(n^2)` and the other one (`B`) is `O(log n)`, it is not said that `A` is slower than `B`. Actually, with 100 items, `A` might be ten times faster than `B`. It only says that with 200 items, `A` will grow slower by the factor `n^2` and `B` will grow slower by the factor `log n`. So, if you benchmark both and you know how much time `A` takes to process 100 items, and how much time `B` needs for

the same 100 items, and `A` is faster than `B`, you can calculate at what amount of items `B` will overtake `A` in speed (as the speed of `B` decreases much slower than the one of `A`, it will overtake `A` sooner or later—this is for sure).

Share Improve this answer

Follow

edited Sep 18, 2013 at 18:22

Gabriel Nahmias

980 ● 3 ● 15 ● 20

answered Sep 25, 2008 at 12:57

Mecki

133k ● 34 ● 259 ● 269

10

Big O notation denotes the limiting factor of an algorithm. Its a simplified expression of how run time of an algorithm scales with relation to the input.

For example (in Java):

```
/** Takes an array of strings and concatenates
them
  * This is a silly way of doing things but it
gets the
  * point across hopefully
  * @param strings the array of strings to
concatenate
  * @returns a string that is a result of the
concatenation of all the strings
  *          in the array
  */
public static String badConcat(String[] Strings){
    String totalString = "";
    for(String s : strings) {
        for(int i = 0; i < s.length(); i++){
            totalString += s.charAt(i);
```

```
        }
    }
    return totalString;
}
```

Now think about what this is actually doing. It is going through every character of input and adding them together. This seems straightforward. The problem is that **String is immutable**. So every time you add a letter onto the string you have to create a new String. To do this you have to copy the values from the old string into the new string and add the new character.

This means you will be copying the first letter *n* times where *n* is the number of characters in the input. You will be copying the character `n-1` times, so in total there will be `(n-1)(n/2)` copies.

This is `(n^2-n)/2` and for Big O notation we use only the highest magnitude factor (usually) and drop any constants that are multiplied by it and we end up with `O(n^2)`.

Using something like a `StringBuilder` will be along the lines of O(nLog(n)). If you calculate the number of characters at the beginning and set the capacity of the `StringBuilder` you can get it to be `O(n)`.

So if we had 1000 characters of input, the first example would perform roughly a million operations, `StringBuilder` would perform 10,000, and the `StringBuilder` with `setCapacity` would perform 1000 operations to do the same thing. This is rough estimate,

but `O(n)` notation is about orders of magnitudes, not exact runtime.

It's not something I use per say on a regular basis. It is, however, constantly in the back of my mind when trying to figure out the best algorithm for doing something.

Share Improve this answer

Follow

edited Oct 14, 2012 at 2:23

liamzebedee

14.5k ● 21 ● 76 ● 118

answered Sep 25, 2008 at 12:51

Steve g

2,489 ● 17 ● 17

5

Every programmer should be aware of what Big O notation is, how it applies for actions with common data structures and algorithms (and thus pick the correct DS and algorithm for the problem they are solving), and how to calculate it for their own algorithms.

1) It's an order of measurement of the efficiency of an algorithm when working on a data structure.

2) Actions like 'add' / 'sort' / 'remove' can take different amounts of time with different data structures (and algorithms), for example 'add' and 'find' are O(1) for a hashmap, but O(log n) for a binary tree. Sort is O(nlog n) for QuickSort, but O(n^2) for BubbleSort, when dealing with a plain array.

3) Calculations can be done by looking at the loop depth of your algorithm generally. No loops, O(1), loops iterating over all the set (even if they break out at some point) O(n). If the loop halves the search space on each iteration? O(log n). Take the highest O() for a sequence of loops, and multiply the O() when you nest loops.

Yeah, it's more complex than that. If you're really interested get a textbook.

Share Improve this answer Follow

edited Sep 25, 2008 at 17:01

community wiki
2 revs
JeeBee

Ok, what is it, how does it apply to common data structures, and how does one calculate it for their own algorithms? – enobrev Sep 25, 2008 at 12:50

Other people had answered that already. – JeeBee Sep 25, 2008 at 16:55

4

A very similar question has already been asked at Big-O for Eight Year Olds?. Hopefully the answers there will answer your question although the question asker there did have a bit of mathematical knowledge about it all which you may not have so clarify if you need a fuller explanation.

Share Improve this answer

Follow

edited May 23, 2017 at 12:02

Community Bot

1 • 1

answered Sep 25, 2008 at 12:32

Luke Bennett

32.9k • 3 • 32 • 57

3

'Big-O' notation is used to compare the growth rates of two functions of a variable (say n) as n gets very large. If function f grows much more quickly than function g we say that g = O(f) to imply that for large enough n, f will *always* be larger than g up to a scaling factor.

It turns out that this is a very useful idea in computer science and particularly in the analysis of algorithms, because we are often precisely concerned with the growth rates of functions which represent, for example, the time taken by two different algorithms. Very coarsely, we can determine that an algorithm with run-time t1(n) is more efficient than an algorithm with run-time t2(n) if t1 = O(t2) for large enough n which is typically the 'size' of the problem - like the length of the array or number of nodes in the graph or whatever.

This stipulation, that n gets large enough, allows us to pull a lot of useful tricks. Perhaps the most often used one is that you can simplify functions down to their fastest growing terms. For example n^2 + n = O(n^2) because as n gets large enough, the n^2 term gets *so much larger*

than n that the n term is practically insignificant. So we can drop it from consideration.

However, it does mean that big-O notation is less useful for small n, because the slower growing terms that we've forgotten about are still significant enough to affect the run-time.

What we now have is a tool for comparing the costs of two different algorithms, and a shorthand for saying that one is quicker or slower than the other. Big-O notation can be abused which is a shame as it is imprecise enough already! There are equivalent terms for saying that a function grows less quickly than another, and that two functions grow at the same rate.

Oh, and do I use it? Yes, all the time - when I'm figuring out how efficient my code is it gives a great 'back-of-the-envelope- approximation to the cost.

Share Improve this answer

Follow

answered Sep 26, 2008 at 16:35

HenryR

8,509 ● 7 ● 36 ● 39

3

The "Intuitition" behind Big-O

Imagine a "competition" between two functions over x, as x approaches infinity: f(x) and g(x).

Now, if from some point on (some x) one function always has a higher value then the other, then let's call this

function "faster" than the other.

So, for example, if for every x > 100 you see that f(x) > g(x), then f(x) is "faster" than g(x).

In this case we would say g(x) = O(f(x)). f(x) poses a sort of "speed limit" of sorts for g(x), since eventually it passes it and leaves it behind for good.

This isn't exactly the definition of big-O notation, which also states that f(x) only has to be larger than C*g(x) for some constant C (which is just another way of saying that you can't help g(x) win the competition by multiplying it by a constant factor - f(x) will always win in the end). The formal definition also uses absolute values. But I hope I managed to make it intuitive.

Share Improve this answer

Follow

answered Jun 3, 2009 at 4:34

Assaf Lavie

75.7k ● 35 ● 150 ● 205

2

It may also be worth considering that the complexity of many algorithms is based on more than one variable, particularly in multi-dimensional problems. For example, I recently had to write an algorithm for the following. Given a set of n points, and m polygons, extract all the points that lie in any of the polygons. The complexity is based around two known variables, n and m, and the unknown of how many points are in each polygon. The big O notation here is quite a bit more involved than O(f(n)) or even O(f(n) + g(m)). Big O is good when you are dealing

with large numbers of homogenous items, but don't expect this to always be the case.

It is also worth noting that the actual number of iterations over the data is often dependent on the data. Quicksort is usually quick, but give it presorted data and it slows down. My points and polygons alogorithm ended up quite fast, close to O(n + (m log(m)), based on prior knowledge of how the data was likely to be organised and the relative sizes of n and m. It would fall down badly on randomly organised data of different relative sizes.

A final thing to consider is that there is often a direct trade off between the speed of an algorithm and the amount of space it uses. Pigeon hole sorting is a pretty good example of this. Going back to my points and polygons, lets say that all my polygons were simple and quick to draw, and I could draw them filled on screen, say in blue, in a fixed amount of time each. So if I draw my m polygons on a black screen it would take O(m) time. To check if any of my n points was in a polygon, I simply check whether the pixel at that point is green or black. So the check is O(n), and the total analysis is O(m + n). Downside of course is that I need near infinite storage if I'm dealing with real world coordinates to millimeter accuracy.... ...ho hum.

answered Sep 27, 2008 at 16:06

SmacL

22.9k ● 15 ● 99 ● 151

2

It may also be worth considering *amortized* time, rather than just worst case. This means, for example, that if you run the algorithm *n* times, it will be *O(1)* on average, but it might be worse sometimes.

A good example is a dynamic table, which is basically an array that expands as you add elements to it. A naïve implementation would increase the array's size by 1 for each element added, meaning that all the elements need to be copied every time a new one is added. This would result in a $O(n^2)$ algorithm if you were concatenating a series of arrays using this method. An alternative is to double the capacity of the array every time you need more storage. Even though appending is an *O(n)* operation sometimes, you will only need to copy *O(n)* elements for every *n* elements added, so the operation is *O(1)* on average. This is how things like *StringBuilder* or *std::vector* are implemented.

Share Improve this answer

Follow

answered Sep 27, 2008 at 16:19

Jay Conrod

29.6k ● 20 ● 99 ● 110

2

What is Big O notation?

Big O notation is a method of expressing the relationship between many steps an algorithm will require related to the size of the input data. This is referred to as the algorithmic complexity. For example sorting a list of size N using Bubble Sort takes O(N^2) steps.

Do I use Big O notation?

I do use Big O notation on occasion to convey algorithmic complexity to fellow programmers. I use the underlying theory (e.g. Big O analysis techniques) all of the time when I think about what algorithms to use.

Concrete Examples?

I have used the theory of complexity analysis to create algorithms for efficient stack data structures which require no memory reallocation, and which support average time of O(N) for indexing. I have used Big O notation to explain the algorithm to other people. I have also used complexity analysis to understand when linear time sorting O(N) is possible.

Share Improve this answer

Follow

answered Oct 18, 2009 at 14:27

cdiggins

18.2k ● 7 ● 109 ● 107

1

From Wikipedia.....

Big O notation is useful when analyzing algorithms for efficiency. For example, the time (or the number of steps) it takes to complete a problem of size n might be found to be $T(n) = 4n^2 - 2n + 2$.

As n grows large, the n^2 term will come to dominate, so that all other terms can be neglected — for instance when n = 500, the term $4n^2$ is 1000 times as large as the 2n term. Ignoring the latter would have negligible effect on the expression's value for most purposes.

Obviously I have never used it..

Share Improve this answer

Follow

answered Sep 25, 2008 at 12:34

Brian G

54.9k ● 58 ● 127 ● 141

1

You should be able to evaluate an algorithm's complexity. This combined with a knowledge of how many elements it will take can help you to determine if it is ill suited for its task.

Share Improve this answer

Follow

answered Sep 25, 2008 at 12:39

Bernard

45.5k ● 18 ● 56 ● 70

0

It says how many iterations an algorithm has in the worst case.

to search for an item in an list, you can traverse the list until you got the item. In the worst case, the item is in the last place.

Lets say there are n items in the list. In the worst case you take n iterations. In the Big O notiation it is O(n).

It says factualy how efficient an algorithm is.

Share Improve this answer

Follow

answered Sep 25, 2008 at 12:34

Ikke
101k ● 23 ● 100 ● 119

That's wrong IMHO. Big-O means a complexity class in general. Whether it is worst case, average case or best case has to be added for clarity. – Thorsten79 Sep 25, 2008 at 12:50

Also, Big-O simply says that an algorithm's cost is *no worse* than a given function, it says nothing about how tight that bound is. – HenryR Sep 27, 2008 at 16:24