





Computing > Computer science > Algorithms > Asymptotic notation  
Asymptotic notation


 Asymptotic notation


 Big- $\theta$  (Big-Theta) notation

 Functions in asymptotic notation

 Practice: Comparing function growth

 Big-O notation

 Big- $\Omega$  (Big-Omega) notation

 Practice: Asymptotic notation

< Computing · Computer science · Algorithms · Asymptotic notation

## Big- $\theta$ (Big-Theta) notation

 Google Classroom  Facebook  Twitter  Email

Let's look at a simple implementation of linear search:

```
var doLinearSearch = function(array, targetValue) {  
  for (var guess = 0; guess < array.length; guess++) {  
    if (array[guess] === targetValue) {  
      return guess; // found it!  
    }  
  }  
  return -1; // didn't find it  
};
```

Let's denote the size of the array (`array.length`) by  $n$ . The maximum number of times that the for-loop can run is  $n$ , and this worst case occurs when the value being searched for is not present in the array.

Each time the for-loop iterates, it has to do several things:

- compare `guess` with `array.length`
- compare `array[guess]` with `targetValue`
- possibly return the value of `guess`
- increment `guess`.

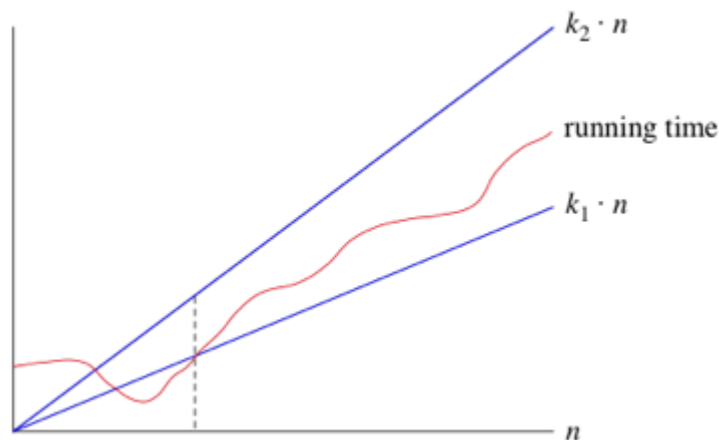
Each of these little computations takes a constant amount of time each time it executes. If the for-loop iterates  $n$  times, then the time for all  $n$  iterations is  $c_1 \cdot n$ , where  $c_1$  is the sum of the times for the computations in one loop iteration. Now, we cannot say here what the value of  $c_1$  is, because it depends on the speed of the computer, the programming language used, the compiler or interpreter that translates the source program into runnable code, and other factors.

This code has a little bit of extra overhead, for setting up the for-loop (including initializing `guess` to 0) and possibly returning `-1` at the end. Let's call the time for this overhead  $c_2$ , which is also a constant. Therefore, the total time for linear search in the worst case is  $c_1 \cdot n + c_2$ .

As we've argued, the constant factor  $c_1$  and the low-order term  $c_2$  don't tell us about the rate of growth of the running time. What's significant is that the worst-case running time of linear search grows like the array size  $n$ . The

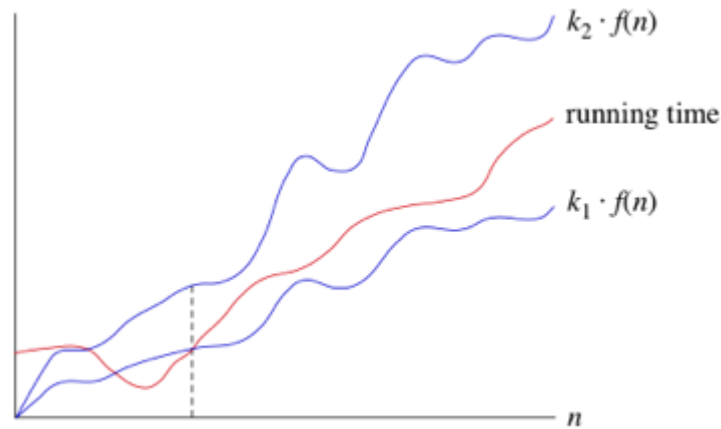
notation we use for this running time is  $\Theta(n)$ . That's the Greek letter "theta," and we say "big-Theta of  $n$ " or just "Theta of  $n$ ."

When we say that a particular running time is  $\Theta(n)$ , we're saying that once  $n$  gets large enough, the running time is at least  $k_1 \cdot n$  and at most  $k_2 \cdot n$  for some constants  $k_1$  and  $k_2$ . Here's how to think of  $\Theta(n)$ :



For small values of  $n$ , we don't care how the running time compares with  $k_1 \cdot n$  or  $k_2 \cdot n$ . But once  $n$  gets large enough—on or to the right of the dashed line—the running time must be sandwiched between  $k_1 \cdot n$  and  $k_2 \cdot n$ . As long as these constants  $k_1$  and  $k_2$  exist, we say that the running time is  $\Theta(n)$ .

We are not restricted to just  $n$  in big- $\Theta$  notation. We can use any function, such as  $n^2$ ,  $n \log_2 n$ , or any other function of  $n$ . Here's how to think of a running time that is  $\Theta(f(n))$  for some function  $f(n)$ :



Once  $n$  gets large enough, the running time is between  $k_1 \cdot f(n)$  and  $k_2 \cdot f(n)$ .

In practice, we just drop constant factors and low-order terms. Another advantage of using big- $\Theta$  notation is that we don't have to worry about which time units we're using. For example, suppose that you calculate that a running time is  $6n^2 + 100n + 300$  microseconds. Or maybe it's milliseconds. When you use big- $\Theta$  notation, you don't say. You also drop the factor 6 and the low-order terms  $100n + 300$ , and you just say that the running time is  $\Theta(n^2)$ .

When we use big- $\Theta$  notation, we're saying that we have an **asymptotically tight bound** on the running time. "Asymptotically" because it matters for only large values of  $n$ . "Tight bound" because we've nailed the running time to within a constant factor above and below.

This content is a collaboration of [Dartmouth Computer Science](#) professors [Thomas Cormen](#) and [Devin Balkcom](#), plus the Khan Academy computing curriculum team. The content is licensed [CC-BY-NC-SA](#).

Sort by: Top Voted ▼

[Questions](#)

Tips & Thanks

## Want to join the conversation?

You need at least 5000 energy points to get started.



**Jessica Sachs** 5 years ago



more ▼

I was confused about where  $k_1$  and  $k_2$  came from, as well as tight bound. I read this StackOverflow answer and it gave me a bit more to go on: <http://stackoverflow.com/a/471292/17188> -- anyone else have a better explanation/clarification?

2 comments

(129 votes) Flag more ▼



**Noble Mushtak** 5 years ago



▼



$k_1$  and  $k_2$  are simply real numbers that could be anything as long as  $f(n)$  is between  $k_1 \cdot f(n)$  and  $k_2 \cdot f(n)$ .

Let's say that `doLinearSearch(array, targetValue)` runs at  $f(n) = 2n + 3$  speed in microseconds on a certain computer (where  $n$  is the length of the array) and we're trying to prove that it has  $\Theta(n)$  time complexity. We would need to find two real numbers  $k_1$ ,  $k_2$ , and  $n_0$  such that  $k_1 \cdot n < 2n + 3 < k_2 \cdot n$  for all  $n > n_0$ . We need to solve for  $k_1$ ,  $k_2$ , and  $n_0$ . It happens that there are infinitely many solutions, but we only need to find one to complete our proof.

Now, it happens that the solution  $k_1 = 1$ ,  $k_2 = 3$ , and  $n_0 = 4$  works. This means  $1 \cdot n < 2n + 3 < 3n$  for all  $n > 4$ . For example:

$$1 \cdot 5 < 2 \cdot 5 + 3 < 3 \cdot 5 \rightarrow 5 < 13 < 15$$

$$1 \cdot 6 < 2 \cdot 6 + 3 < 3 \cdot 6 \rightarrow 6 < 15 < 18$$

$$1 \cdot 7 < 2 \cdot 7 + 3 < 3 \cdot 7 \rightarrow 7 < 17 < 21$$

$$1 \cdot 4.1 < 2 \cdot 4.1 + 3 < 3 \cdot 4.1 \rightarrow 4.1 < 11.2 < 12.3$$

In mathematics, these examples alone wouldn't actually suffice for a proof, but there is a rigorous way to prove that this inequality is true. It's just rather complicated.

You might ask why I didn't pick  $n_0 = 3$  instead. After all, that seems to be the minimal possible answer, because 3 doesn't work ( $1 \cdot 3 < 2 \cdot 3 + 3 < 3 \cdot 3 \rightarrow 3 < 9 < 9$ ), but all  $n > 3$  works. However, while that's true, I don't need the minimal possible  $n_0$ . I just need *any*  $n_0$  that works. If I find one solution, then I've proved what I've needed to.

I would imagine if you were taking a college level computer science course, you might need need to prove Big- $\Theta$ , Big- $O$ , and Big- $\Omega$  statements. Also, the make-up of the actual function would probably be determined by something else because you don't usually work with actual microseconds or milliseconds in general time complexity. For

example, maybe the algorithm has two loops that go from 0 to  $n$  (where  $n$  is the input of the algorithm), in which case the function might be  $f(n)=2n$ . In general, you won't need to rigorously prove any complexity theory statements because as I said before, it's rather complicated (although won't be if you have experience proving limits). Instead, you can just keep the rule of thumb for polynomials that you can drop all of the terms except the one with the highest exponent, and then drop the exponent.

I hope this helps!

7 comments

(164 votes)  Flag [more](#) 

See 14 more replies



**Jim Collings** 4 years ago



[more](#) 

So it seems to me that Big O is the worst case scenario, Big Theta is the range of cases from best to worst and Big Omega I haven't got to yet. ...now why couldn't you guys just say that?

1 comment

(11 votes)  Flag [more](#) 



**Cameron** 4 years ago



[more](#) 

They didn't say that, because it would be incorrect.  
Big O, Big Theta and Big Omega are often all used to measure the worst case scenario (this is most often the case).  
They could also all be used to measure an average case (this happens often).  
They could also all be used to measure the best case (we rarely care

about best cases)

The measurements of Big-O, Big-Theta, and Big-Omega would often be different depending on which case was picked.

Here's the simple version of what Big-O, Big-Theta, and Big-Omega are :

If you have a function  $f(N)$ :

Big-O tells you which functions grow at a rate  $\geq$  than  $f(N)$ , for large  $N$

Big-Theta tells you which functions grow at the same rate as  $f(N)$ , for large  $N$

Big-Omega tells you which functions grow at a rate  $\leq$  than  $f(N)$ , for large  $N$

(Note:  $\geq$  , "the same", and  $\leq$  are not really accurate here, but the concepts we use in asymptotic notation are similar):

We often call Big-O an upper bound, Big-Omega a lower bound, and Big-Theta a tight bound. Often in computer science the function we are concerned with is the running time of an algorithm for inputs of size  $N$ .

Hope this makes sense

3 comments

(100 votes)  Flag [more](#) 

See 5 more replies



Freya Murphy 4 years ago



[more](#) 

What exactly does asymptotic mean? In maths at school I learned that an asymptote is a line that a curve never touches but the distance between them approaches zero. However, that definition isn't really helping me to understand this concept.

1 comment

(13 votes)  Flag [more](#) 



**Cameron** 4 years ago

more ▾

When we are talking about "asymptotic" behavior here, we are talking about the behavior of functions when  $n$  is very large (approaching infinity).

We have 3 curves that we worry about.

A curve for the running time function  $f(n)$ . (we often don't know exactly what this function looks like)

A curve that we know is "above" the running time function when  $n$  is large. ( Big-O )

A curve that we know is "below" the running time function when  $n$  is large. (Big-Omega)

(Note: the terms "above" and "below" aren't entirely accurate here, and Big-O and Big-Omega actually describe a set of curves rather than just one curve)

If we can squeeze the curves that are "above" and "below" the running time function close enough, then we can figure out Big-Theta. This is useful when we don't know exactly what the running time curve looks like, which is most often the case, because Big-Theta is just a "scaled" version of the running time curve (it can tell us the shape of the curve). (Note: this isn't entirely accurate, it only scales the highest order behavior of  $f(n)$  (the parts that dominate as  $n$  increases) , and Big-Theta is also a set of curves)

Hope this makes sense

**4 comments**(39 votes)  Flag more ▾

See 1 more reply

**James Macak** 4 years ago[more](#) ▾

I am just wondering what level is this course, I am 15 and it seems to be sorta confusing. I am trying hard to understand the Big-Theta notation, but it seems to get confusing rather quickly. I might just be confusing things in my head, not sure. I am going to continue working on this, but I would like to know what level this course is. Thanks

(13 votes)  [Flag](#) [more](#) ▾**C C** 4 years ago[more](#) ▾

It is difficult to give a grade level to this type of material since when and how it is taught varies so widely. Personally, I think it leans towards college level, especially the asymptotic notation calculations. It is important to know what types of things will slow your programs to the point of frustration. For the amateur coder, it is not so important to calculate and graph it! I don't think most software engineers approach issues that way, unless they need to pitch a project or they are really stuck on a difficult bottleneck.

People who are more recently in CS may disagree (I moved over to math), but I think the much more interesting aspects of algorithm study involve problem solving and translating solutions into code. And until you do a bunch of that, measuring how fast your subroutine goes won't be of much help.

Anyway, hope this is useful and you enjoy whichever areas you end up focusing on.

**1 comment**(23 votes)  [Flag](#) [more](#) ▾[See 4 more replies](#)

**Eduardo Lopes** 4 years ago[more](#) ▾

Where I can find a good reading about how to compute the complexity of an algorithm? I want to understand how to figure out if an algorithm is  $n \log n$ ,  $n^2$ , etc.

(8 votes)  Flag [more](#) ▾**Cameron** 4 years ago[more](#) ▾

Here's the quick version of how to analyze running times:  
(I use  $n$  below for the problem size )

-Replace any simple block of operations that are performed without calling a function with a constant  
e.g.

```
x = 5* 4 + 2;  
y++;  
z = x * y;
```

will run in some constant amount of time  
we usually label that time with some variable e.g. 'c1'

-If you have a for loop that loops  $n$  times, the running time is about  $n$  times the running time of the code inside the loop (this neglects the constant amount of time to initialize the for loop and the constant amount of time every loop to perform the comparison and increment)  
e.g.

```
for(var i=0; i < n ; i++){  
    x = 5* 4 + 2;  
    y++;  
    z = x * y;  
}
```

we determined above that the code inside of the for loop runs in ' $c1$ '  
so the running time of this code is about ' $c1 * n$ '

e.g.

```
for(var j=0; j < n; j++){  
    for(var i=0; i < n ; i++){  
        x = 5* 4 + 2;  
        y++;  
        z = x * y;  
    }  
}
```

we determined above that the code inside of our outermost for loop  
runs in about ' $c1 * n$ '  
so this code runs in about ' $c1 * n * n$ ' or ' $c1 * n^2$ '

-If after each iteration of an algorithm the problem is ' $1/a$ ' of the size  
before the iteration then the code will iterate about ' $\log_a(n)$ ' times.  
Thus, the running time of the code will be about  $\log_a(n)$  times the  
running time of the code run each iteration.

e.g.

```
while( n>1 ){  
    x = 5* 4 + 2;  
    y++;  
    z = x * y;  
  
    n = n/2;  
}
```

the code inside of the while loop runs in some constant time ' $c_2$ '  
the size of the problem size  $n$ , is reduce in  $1/2$  each iteration  
so this code iterates about ' $\log_2(n)$ ' times  
so the running time of the code is about ' $c_2 * \log_2(n)$ '

Beyond the above, I would look at the analysis in the articles here, and study how they find the running times to learn other techniques on how to figure out the running times of algorithms.

"Discrete Mathematics with Applications" by Susanna Epp, discusses how to analyze running times in the chapter on Efficiency of Algorithms.

Hope this makes sense

3 comments

(26 votes)  Flag [more](#) 

See 1 more reply



**Joss Glenn** 4 years ago

[more](#) 

I'm still not clear about what "tightly bound

" means in this context. I do, however understand everything else, including the definition of "Asymptotic".

(6 votes)  Flag [more](#) 



**Cameron** 4 years ago



[more](#) 

The upper bound tells us what grows asymptotically faster than or at the same rate as our function. The lower bound tells us what asymptotically grows slower than or at the same rate as our function. Our function must lie somewhere in between the upper and lower bound.

Suppose that we can squeeze the lower bound and our upper bound closer and closer together. Eventually they will both be at the same asymptotic growth rate as our function. That's what we would call tight bounds (where the upper bound and lower bound have the same growth rate as the function).

(16 votes)  Flag [more](#) 

See 1 more reply



**Motty Waldner** 4 years ago

[more](#) 

I did not understand what you meant when you spoke about the constants  $k_1$  and  $k_2$ ; can you please elaborate some more?

(4 votes)  Flag [more](#) 



**Cameron** 4 years ago

[more](#) 

Suppose you have two functions,  $f(n)$  and  $g(n)$ .

If, for large values of  $n$ , you are able to:

-squeeze  $f(n)$  between  $k_1 * g(n)$  and  $k_2 * g(n)$

Then:

-you can say  $f(n)$  is  $\Theta(g(n))$

You are allowed to pick any value you want for  $k_1$  and  $k_2$  as long as they are both greater than zero.

e.g.

Suppose:  $f(n) = 6n + 5$  and  $g(n) = n + 2$

We could pick  $k_1 = 1$  and  $k_2 = 6$  and  $n \geq 1$

We can see that:  $k_1 * g(n) \leq f(n) \leq k_2 * g(n)$

because:  $1 * (n+2) \leq 6n+5 \leq 6 * (n+2)$

We successfully squeezed  $f(n)$  between  $k_1 * g(n)$  and  $k_2 * g(n)$  so we can say:

$f(n)$  is  $\Theta(g(n))$

Hope this makes sense

2 comments

(5 votes)  Flag [more](#) 

See 1 more reply



**Vijayenthiran Subramaniam** 5 years ago

[more](#) 

In the graph why  $\Theta(n)$  is not linear? the time taken for an increment in the for loop will be a constant always or will it be varying? and in the second graph why  $k_1$  and  $k_2$  are not constant? why it is curved?

is  $\Theta(n) = c_1 \cdot n + c_2$

1 comment

(4 votes)  Flag [more](#) 



**jdsutton** 5 years ago

[more](#) 

That's not a graph of  $\Theta(n)$ , it's a graph of the running time.  $\Theta(n)$  represents the upper and lower bound on the running time.

4 comments

(3 votes)  Flag [more](#) 

See 2 more replies



**barggros** 4 years ago

[more](#) 

Ok I have a question, if for example I have  $(n+1)n/2$  and im trying to prove that is not a set of  $\Theta(n)$ , but I am a bit confused, so say for example  $k_1, k_2$  and  $N$ , does  $k_1$  always have to be 1 or can it be greater than one, say 2, and  $k_1=2, k_2=4$  and  $N=1$ : then by doing  $2 \cdot 2 \leq (2+1)2/2 \leq 4 \cdot 2$ , which is equivalent to  $4 \leq 3 \leq 8$ , by my understanding this would prove it false but, as you may know I am still confused and would like to know if the constant  $k_1$  always have to be 1 or can it be greater?

(4 votes)  Flag [more](#) 



**Leonardo Hernández Men...** 4 years ago

[more](#) 

Now im even more confused, what do you mean divide everything by  $n$ ?  
Give me an example please



(2 votes)  Flag [more](#) [See 1 more reply](#)**Matthias Schmitz** 5 years ago[more](#) 

In the first paragraph, why is "possibly return the value of guess" included in the list of things that happens each time the for loop iterates? Shouldn't it be included in c2 instead because it only happens once?

**1 comment**(3 votes)  Flag [more](#) **Michael Krebs** 4 years ago[more](#) 

The reason why this is included in c1 rather than c2 is a little surprising. While returning the value of the guess occurs at most once, *not returning the value of the guess occurs each time the loop iterates*. An "if" statement must skip past the code that will not be run when the condition is false, and the jump over the code that won't happen is what takes up this small, but very real, amount of time each iteration.

(7 votes)  Flag [more](#) [See 2 more replies](#)[Show more...](#)

[◀ Asymptotic notation](#)

[Functions in asymptotic notation ▶](#)