



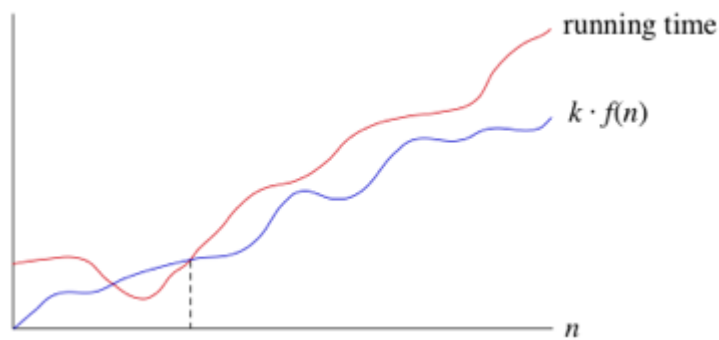
< Computing • Computer science • Algorithms • Asymptotic notation

Big- Ω (Big-Omega) notation








 Google Classroom  Facebook  Twitter  Email

Sometimes, we want to say that an algorithm takes *at least* a certain amount of time, without providing an upper bound. We use big- Ω notation; that's the Greek letter "omega."

If a running time is $\Omega(f(n))$, then for large enough n , the running time is at least $k \cdot f(n)$ for some constant k . Here's how to think of a running time that is $\Omega(f(n))$:



Computing > Computer science > Algorithms > Asymptotic notation
Asymptotic notation

-  Asymptotic notation
-  Big- θ (Big-Theta) notation
-  Functions in asymptotic notation
-  Practice: Comparing function growth
-  Big-O notation
-  Big- Ω (Big-Omega) notation
-  Practice: Asymptotic notation

We say that the running time is "big- Ω of $f(n)$." We use big- Ω notation for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.

Just as $\Theta(f(n))$ automatically implies $O(f(n))$, it also automatically implies $\Omega(f(n))$. So we can say that the worst-case running time of binary search is $\Omega(\log_2 n)$.

We can also make correct, but imprecise, statements using big- Ω notation. For example, if you really do have a million dollars in your pocket, you can truthfully say "I have an amount of money in my pocket, and it's **at least** 10 dollars." That is correct, but certainly not very precise. Similarly, we can correctly but imprecisely say that the worst-case running time of binary search is $\Omega(1)$, because we know that it takes **at least** constant time.

Of course, typically, when we are talking about algorithms, we try to describe their running time as precisely as possible. We provide the examples of the imprecise statements here to help you better understand big- Ω , big- O , and big- Θ .

This content is a collaboration of [Dartmouth Computer Science](#) professors [Thomas Cormen](#) and [Devin Balkcom](#), plus the Khan Academy computing

[Next lesson](#)[Selection sort](#)

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.



Georgiy Ivankin 5 years ago

[more](#) ▾

I can't wrap my head around this: why it is correct to say that the for the binary search the running time is $\Omega(\lg n)$?

The 1st paragraph clearly says that Ω defines the *least* running time, and for the binary search the least running time is 1 guess, no matter how big is n, no?

The statement that $\Theta(f(n))$ automatically implies $\Omega(f(n))$ seems incomprehensible to me... What do I get wrong?

3 comments

(88 votes) Flag [more](#) ▾



Cameron 5 years ago



more ▾

When we use asymptotic notation, unless stated otherwise, we are talking about **worst-case running time**.

The worst-case running time for binary search is $\log(n)$.

Recall:

if $f(n)$ is $O(g(n))$ this means that $f(n)$ grows asymptotically no faster than $g(n)$

if $f(n)$ is $\Omega(g(n))$ this means that $f(n)$ grows asymptotically no slower than $g(n)$

if $f(n)$ is $\Theta(g(n))$ this means that $f(n)$ grows asymptotically at the same rate as $g(n)$

As a result:

if $f(n)$ is $\Theta(g(n))$ it is growing asymptotically at the same rate as $g(n)$. So we can say that $f(n)$ is not growing asymptotically slower or faster than $g(n)$. But from the above, we can see this means that $f(n)$ is $\Omega(g(n))$ and $f(n)$ is $O(g(n))$.

Think of O as an upperbound, and Ω as a lower bound.

These bounds can be tight or loose, but we prefer to make them tight as possible.

If we have tight bounds where O and Ω have the same growth rate then we precisely know the growth rate.

If we can precisely give the growth rate then we know Θ .

An analogy is the height of a neighbour.

We can immediately say that the height of our neighbour is upper bounded by 1 million kilometers.

We can also say that the height of our neighbour is lower bounded by 1 nanometer.

These statements aren't very useful, because these bounds are so loose. However if we gave a lower bound for the height of our neighbour at 5' 5", and an upper bound of 5' 10" we would have a much better idea of

how tall our neighbour was.

If we had a lower bound of 5' 8" and an upper bound of 5' 8" we could then say that our neighbour is 5' 8".

So for $\log(n)$ we could say:

$\log(n)$ is $O(n^n)$ since $\log(n)$ grows asymptotically no faster than n^n

$\log(n)$ is $O(n)$ since $\log(n)$ grows asymptotically no faster than n

$\log(n)$ is $O(\log(n))$ since $\log(n)$ grows asymptotically no faster than $\log(n)$

We went from loose upper bounds to a tight upper bound

$\log(n)$ is $\Omega(1)$ since $\log(n)$ grows asymptotically no slower than 1

$\log(n)$ is $\Omega(\log(\log(n)))$ since $\log(n)$ grows asymptotically no slower than $\log(\log(n))$

$\log(n)$ is $\Omega(\log(n))$ since $\log(n)$ grows asymptotically no slower than $\log(n)$

We went from loose lower bounds to a tight lower bound

Since we have $\log(n)$ is $O(\log(n))$ and $\Omega(\log(n))$ we can say that $\log(n)$ is $\Theta(\log(n))$.

Hope this makes sense

24 comments

(333 votes)  Flag [more](#) 

See 7 more replies



John Phipps 5 years ago



[more](#) 

If we know the big-Theta of a function, is there a reason that we would want to know a different big-O or big-Omega for the function? It just appears that if you have the big-Theta you would know the best big-O and big-Omega so I don't know why you would want to know these values unless they were significantly

easier to find than the big-Theta. Does the big-Theta give the best big-O and big-Omega possible?

(15 votes)  Flag [more](#) 



Cameron 5 years ago



[more](#) 

Typically, if we design an algorithm and perform some quick analysis on it we find a reasonable O value (it doesn't have to be tight, but it shouldn't be too loose).

Often, that's all we care about.

Sometimes, we then find a reasonable Ω value (again it doesn't have to be tight, but it shouldn't be too loose).

Then, if it is critical to have tight bounds, or if the algorithm is easy to analyze we tighten up the bounds on O and Ω until they are the same and then we have Θ .

A typical scenario is: we have some task that our program needs to perform.

Often, we don't care about which algorithm we use as long as the asymptotic complexity of the algorithm is under some limit i.e. the algorithm is fast enough for our needs.


If we write an algorithm that gets the job done, but it is hard to analyze, we can often make simplifying assumptions that make the analysis easier but the bounds looser.

e.g.

Suppose I had some code that looked like this:

```
for(var j=1; j<=n;j++){  
    var number= some_hard_to_analyze_calculation
```

```
    if (number < 1){ number=1;}
    if( number > n){ number=n;}
    for(var k=1; k <=number;k++){
        do_some_simple_stuff();
    }
}
```



I could quickly see that the outer for loop loops n times.

I can see that, just before the inner for loop, number is some value between 1 and n .

So the inner for loop, loops at most n times.

Since $\text{outer_loops} * \text{inner_loops} = n * n = n^2$, I could say that the running time is $O(n^2)$

If I only cared that the algorithm runs no slower than $O(n^2)$ I could stop analyzing.

Suppose I was also concerned about the lower bound (we often don't care about the lower bound as much),

I can see that the inner for loop loops at least 1 time.

Since $\text{outer_loops} * \text{inner_loops} = n * 1 = n$, I could say that the running time is $\Omega(n)$

So now I would know that this algorithm has a running time that is $\Omega(n)$ and $O(n^2)$

Since the calculation is hard to analyze I might not be able to tighten up Ω or O .

If I can't tighten up the bounds on Ω or O , I can't figure out Θ .

Hope this makes sense

7 comments

(56 votes)  Flag [more](#) [See 4 more replies](#)**Salman Oskooi** 4 years ago[more](#) 

My question has already been asked below, and only answered by Cameron, whose explanations I'm not understanding. Stephen Henn said: "How can that be: 'So we can say that the worst-case running time of binary search is $\Omega(\lg n)$.' If Ω is the time an algorithm takes at the minimal running time?" Also jamie_chu78 asked the same question and got the same answer. Is there anyone who can explain in a different manner how Ω , which is the lower bound of running time, can be the worst-case running time? It really does seem like O should be the worst-case since that is the upper bound, or maximum amount of running time that $f(n)$ can take.

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

Perhaps this will clear things up.

A GAME

Suppose the following:

You are trapped in a prison and the warden of the prison will only let you go after you play his game.

He shows you two identical looking boxes and tells you:

- One box has between 10 and 20 bugs in it (we'll call this Box A)
- One box has between 30 and 40 bugs in it (we'll call this Box B)

You have to pick one of the boxes and eat the bugs inside the box.
You don't know which box is Box A or which box is Box B.

The warden knows, but doesn't tell you, that Box A actually has 17 bugs in it, and Box B actually has 32 bugs in it.

WHAT WE MEAN BY UPPER AND LOWER BOUND

Let's clarify what we mean by upper bound and lower bound by using Box A as an example.

A lower bound for X is a value that X can not be below.

So, knowing that Box A has between 10 and 20 bugs we can say that:
-Box A can not have less than 5 bugs (this should be obvious, since it is less than 10)

This means that we can say that 5 is a lower bound on the number of bugs in Box A.

We can also say that:

-Box A can not have less than 10 bugs (this should be obvious, since it is equal to 10)

This means that we can say that 10 is a lower bound on the number of bugs in Box A.

Both 5 and 10 are valid lower bounds for the number of bugs in Box A.

However, the lower bound of 10 is much more useful than the lower bound of 5, because it is closer to the actual number of bugs in the box. We tend to only mention the lower bound that is closest to the actual value, because it is the most useful.

Similarly, an upper bound for X is a value that X can not be above.

We can say that:

-Box A can not have more than 50 bugs (this should be obvious, since it is more than 20)

This means that we can say that 50 is an upper bound on the number of bugs in Box A.

We can also say that:

-Box A can not have more than 20 bugs (this should be obvious, since it is equal to 20)

This means that we can say that 20 is an upper bound on the number of bugs in Box A.

Both 20 and 50 are valid upper bounds for the number of bugs in Box A.

However, the upper bound of 20 is much more useful than the upper bound of 50, because it is closer to the actual number of bugs in the box. We tend to only mention the upper bound that is closest to the actual value, because it is the most useful.

The actual value of X must always be between the lower bound of X and the upper bound of X .

For Box A the actual number of bugs in Box A must be between our lower and upper bounds for the number of bug in Box A. (This is true since 17 is between 10 and 20)

ANALYZING UPPER AND LOWER BOUNDS IN THE BEST AND WORST CASE

You decide to analyze the upper and lower bounds on the number of bugs you have to eat in the best and worst case scenarios.

(Let's assume that eating less bugs is better than eating more bugs)

The best case scenario: you pick Box A

Since Box A has between 10 and 20 bugs in it:

The lower bound on the number of bugs you have to eat, in the best case scenario, is 10

The upper bound on the number of bugs you have to eat, in the best case scenario, is 20

(In the best case, the actual number of bugs you have to eat is 17, but you don't know this.)

The worst case scenario: you pick Box B

Since Box B has between 30 and 40 bugs in it:

The lower bound on the number of bugs you have to eat, in the worst case scenario, is 30

The upper bound on the number of bugs you have to eat, in the worst case scenario, is 40

(In the worst case, the actual number of bugs you have to eat is 32, but you don't know this.)

So we can see from the above that:

- the best case scenario has both lower and upper bounds
- the worst case scenario has both lower and upper bounds

What we can say (and what often causes people to confuse lower and upper bounds with best case and worse case) is that:

- the upper bound, in the worst case, is as bad as it can possibly get (the upper bound in the worst case is the absolute upper bound for ALL cases)
- the lower bound, in the best case, is as good as it can possibly get (the lower bound in the best case is the absolute lower bound for ALL cases)

APPLYING THIS TO BINARY SEARCH

So how does this apply to binary search ?

Let's analyze the best case and worst case for binary search.

In the best case binary search finds its target on the first guess.

Analysis shows that the running time, $f(n)$, will be constant in this scenario.

i.e. $f(n) = c_1$ (where c_1 is a constant)

The lower bound on the running time, $f(n)$, in the best case scenario, is $\Omega(1)$

The upper bound on the running time, $f(n)$, in the best case scenario, is $O(1)$

In the worst case binary search doesn't find its target.

Analysis shows that this will require $\sim \log(n)$ guesses.

The running time, $f(n)$, in this scenario, will be:

$f(n) = c_1 * \log(n) + c_2$ (where c_1 and c_2 are constants)

The lower bound on the running time, $f(n)$, in the worst case scenario, is $\Omega(\log(n))$

The upper bound on the running time, $f(n)$, in the worst case scenario, is $O(\log(n))$

It should be mentioned that, often, for complex algorithms, we don't know what $f(n)$ is in each of the scenarios (similar to how we didn't know the actual number of bugs we would eat in each scenario). However, we can often make simplifying assumptions that let us figure out the upper and lower bounds for $f(n)$ in each scenario (similar to how we could figure out upper and lower bounds for the number of bugs we would have to eat for each scenario).

Hope this makes sense

11 comments

(65 votes)  Flag [more](#) 

See 6 more replies


Antoine LeBel 4 years ago

[more](#) ▼

What happens if there's an algorithm where $O(g(n))$ and $\Omega(g(n))$ are not the same? It is even possible? For example having $O(n^2)$ and $\Omega(\lg n)$? If this is the case, is $\Theta(g(n))$ undefined?

Another way to ask : Does $\Theta(g(n))$ only exist if $O(g(n))$ and $\Omega(g(n))$ have the same growth rate?

 (4 votes) Flag [more](#) ▼

Cameron 4 years ago

[more](#) ▼

Suppose I have a really complicated algorithm, which has a running time $f(n)$.

Unfortunately I don't know what $f(n)$ is.

I analyze the algorithm and show that the worst case running time is $\leq 150 \cdot n^2 + 6$

Then I can say $f(n)$ is $O(n^2)$.

I analyze the algorithm again and show that the worst case running time is $\geq 0.1 \cdot \log(n)$

Then I can say $f(n)$ is $\Omega(\log(n))$

At this point, my analysis isn't good enough for me to make a statement about $\Theta(g(n))$

I would only be able to tell you that $f(n)$ is $\Theta(g(n))$ if my analysis showed that $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Suppose that I sharpen my pencil and figure out tighter bounds on the worst case running time of the algorithm. Suppose, I find that:

- $f(n)$ is $\leq 100 \cdot n^{1.5} + 1000$ i.e. $O(n^{1.5})$

- $f(n)$ is $\geq 10 \cdot n^{1.5} + 2$ i.e. $\Omega(n^{1.5})$.

Then, I would be able to say $f(n)$ is $\Theta(n^{1.5})$ since $f(n)$ is $O(n^{1.5})$ and $f(n)$

is $\Omega(n^{1.5})$.

However, if I wasn't able to improve my analysis, and I still only knew that $f(n)$ is $O(n^2)$ and $f(n)$ is $\Omega(\log n)$, I wouldn't be able to give you a statement about $\Theta(g(n))$. However, it is trivially true that $f(n)$ is $O(f(n))$ and that $f(n)$ is $\Omega(f(n))$ and thus $f(n)$ is $\Theta(f(n))$ i.e. we may not be clever enough to find $\Theta(g(n))$, but it exists.

Hope this makes sense

1 comment

(13 votes)  Flag [more](#) 



Andrew Chastain 4 years ago

[more](#) 

Alright...I've bashed my head against this for too long. I am having a really hard time understanding the notation that is introduced in the quiz. For example, if I have $f(n) = n^k$ (polynomial) and $g(n) = c^n$ (exponential) then I would know that for very large n that $g(n) > f(n)$. The only relationships they can have in asymptotic notation is that $g(n)$ is an upper bound of $f(n)$ or $f(n)$ is a lower bound of $g(n)$. I would expect this to be written as $g(n)$ is $O(f(n))$ or $f(n)$ is $\Omega(g(n))$, yet the test indicates $f(n)$ is $O(g(n))$. Could someone please explain why a slower growing function is the upper bound to a faster growing function?

(4 votes)  Flag [more](#) 



Cameron 4 years ago

[more](#) 

Perhaps, this may clear up the notation:

Big Oh, Big Omega and Big Theta are not operators, they are sets.

$O(g(n))$ is the set of all functions that are asymptotically $\leq g(n)$

When we say ' $f(n)$ is $O(g(n))$ ' or ' $f(n) = O(g(n))$ ' what we really mean is:

' $f(n) \in O(g(n))$ ' which means $f(n)$ is a member of the set $O(g(n))$
 i.e. $f(n)$ is in the set of functions that are asymptotically $\leq g(n)$
 Since $f(n)$ is asymptotically $\leq g(n)$, that is why we say $g(n)$ is an upper bound to $f(n)$

Why do people use ' $f(n)$ is $O(g(n))$ ' or ' $f(n) = O(g(n))$ ' instead of saying what we really mean, i.e. ' $f(n) \in O(g(n))$ '?

The use of "is" and "=" was used for so long, and so often, that despite many considering them to be an abuse of the notation, they started to dominate common usage. This is analogous to how the frequent abuse of the word "literally", in recent times, has resulted in dictionaries changing to reflect the new usage.

Hope this makes sense

1 comment

(3 votes)  Flag [more](#) 

See 3 more replies



Srikant Mahapatra 3 years ago

[more](#) 

The last sentence of this article seems to be erroneous: "you can also say that the worst-case running time of binary search is $\Omega(1)$, because it takes at least constant time." Shouldn't it be that the best-case running time of binary search is $\Omega(1)$?

(3 votes)  Flag [more](#) 



Cameron 3 years ago

[more](#) 

Both the best and worst case are $\Omega(1)$.

The running time, in the worst case, will definitely grow faster than constant time, so we can say that it is $\Omega(1)$. However, we can also make the much stronger statement that the running time, in the worst case, is also $\Omega(\lg n)$. Typically, when we are talking about algorithms we only mention the strongest statement we can make, but that does not invalidate the weaker statements.

2 comments

(5 votes)  Flag [more](#) 

See 1 more reply

**Thin** 4 years ago[more](#) 

The statement that $\Theta(f(n))$ automatically implies $\Omega(f(n))$ seems incomprehensible to me...

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

if $f(n)$ is $O(g(n))$ this means that $f(n)$ grows asymptotically no faster than $g(n)$

if $f(n)$ is $\Omega(g(n))$ this means that $f(n)$ grows asymptotically no slower than $g(n)$

if $f(n)$ is $\Theta(g(n))$ this means that $f(n)$ grows asymptotically at the same rate as $g(n)$

If $f(n)$ is $\Theta(g(n))$ then $f(n)$ does not grow slower than $g(n)$, because it grows asymptotically at the same rate as $g(n)$. Since $f(n)$ does not grow slower than $g(n)$, it implies that $f(n)$ is $\Omega(g(n))$.

Similarly, if $f(n)$ is $\Theta(g(n))$ then $f(n)$ does not grow faster than $g(n)$, because it grows asymptotically at the same rate as $g(n)$. Since $f(n)$ does not grow faster than $g(n)$, it implies that $f(n)$ is $O(g(n))$

1 comment

(6 votes)  Flag [more](#) 

See 1 more reply



Hayyan 5 years ago

[more](#) 

Hello!

I could get the idea of the upper and lower bounds, using O and Ω , but I didn't get the following sentence:

"you can also say that the worst-case running time of binary search is $\Omega(1)$, because it takes at least constant time."

as we know that the worst case in binary search for n -size array will always be $\log(n)+1$ tries.

and in the known case, like the worst case, $O(f(n)) = \Omega(f(n)) = \Theta(f(n))$; $f(n) = \ln(n)$;

and the complexity of the binary search **in worst case** will never be constant unless the array contains 1 item, right?

(0 votes)  Flag [more](#) 



Cameron 5 years ago

[more](#) 

if $f(n)$ is $\Omega(g(n))$ this means that $f(n)$ grows asymptotically no slower than $g(n)$

$\log(n)$ is $\Omega(1)$ since $\log(n)$ grows asymptotically no slower than 1

$\log(n)$ is also $\Omega(\log(n))$ since $\log(n)$ grows asymptotically no slower than

$\log(n)$

So, saying that the worst case running time for binary search is $\Omega(1)$, or $\Omega(\log(n))$ are both correct. However, $\Omega(\log(n))$ provides a more useful lower bound than $\Omega(1)$.

2 comments

(6 votes)  Flag [more](#) 

See 2 more replies



justinj1776 4 years ago

[more](#) 

Could anyone tell how we could select $k=5$ as illustrated here: <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/e/quiz--asymptotic-notation> in this quiz.

The thing that confuses me is since $\log_8(n) = \lg(n) / \lg(8) = (1/\lg(8)) * \lg(n)$ is it permissible to take the value of k as greater than 1 while you could see by the equation that it's constant multiplier is $(1/\lg(8))$ which will always produce an value less than 1? Here $1/\lg(8) = 1/3 = 0.33$ (value less than 1)

1 comment

(2 votes)  Flag [more](#) 



Cameron 4 years ago

[more](#) 

I think what you may be missing is that when one says:
 $f(n)$ is $O(g(n))$
 you are really saying:
 for sufficiently large values of n , there exists some constant $k > 0$, such that:
 $f(n) \leq k * g(n)$.

So in the hints, they are picking k to be 5. That is, they are saying:

$$\log(n) \leq 5 * \log_8(n)$$

$$\log(n) \leq 5 * \frac{1}{\log(8)} * \log(n)$$

$$\log(n) \leq 5 * \frac{1}{3} * \log(n)$$

$$\log(n) \leq \frac{5}{3} * \log(n) \text{ (which is obviously true)}$$

Hope this make sense

1 comment

(4 votes)  Flag [more](#) 



   **Natth4545**    5 years ago

[more](#) 

Isn't big omega infinity?

(2 votes)  Flag [more](#) 



jdsutton 5 years ago

[more](#) 

The time it takes to perform some function as n increases may go to infinity. Big omega tells you how quickly that time increases.

(3 votes)  Flag [more](#) 

See 2 more replies

Show more...

[◀ Big-O notation](#)

[Asymptotic notation ▶](#)