

Join the Stack Overflow Community

Stack Overflow is a community of 7.0 million programmers, just like you, helping each other.
Join them; it only takes a minute:

Sign up

What does $O(\log n)$ mean exactly?



Free sandbox.
Hassle free.

Create a web app
No credit card required

I am currently learning about Big O Notation running times and **amortized times**. I understand the notion of $O(n)$ linear time, meaning that the size of the input affects the growth of the algorithm proportionally...and the same goes for, for example, quadratic time $O(n^2)$ etc..even algorithms, such as **permutation generators**, with $O(n!)$ times, that **grow by factorials**.

For example, the following function is $O(n)$ because the algorithm grows in proportion to its input n :

```
f(int n) {
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", i);
}
```

Similarly, if there was a nested loop, the time would be $O(n^2)$.

But what exactly is **$O(\log n)$** ? For example, what does it mean to say that the **height of a complete binary tree is $O(\log n)$** ?

I do know (maybe not in great detail) what Logarithm is, in the sense that: $\log_{10} 100 = 2$, but I cannot understand how to **identify a function with a logarithmic time**.

time-complexity big-o

edited May 14 '16 at 14:53

asked Feb 21 '10 at 20:05



Johan

52.3k

17

118

221



Andreas Grech

55.3k

86

255

331

34 A 1-node binary tree has height $\log_2(1)+1 = 1$, a 2-node tree has height $\log_2(2)+1 = 2$, a 4-node tree has height $\log_2(4)+1 = 3$, and so on. An n -node tree has height $\log_2(n)+1$, so adding nodes to the tree causes its average height to grow logarithmically. – David R Tribble Feb 21 '10 at 22:40

26 One thing I'm seeing in most answers is that they essentially describe "O(something)" means the running time of the algorithm grows in proportion to "something". Given that you asked for "exact meaning" of " $O(\log n)$ ", it's not true. That's the intuitive description of Big-Theta notation, not Big-O. $O(\log n)$ intuitively means the running time grows **at most** proportional to " $\log n$ ": stackoverflow.com/questions/471199/... – Mehrdad Afshari Feb 22 '10 at 10:42

19 I always remember divide and conquer as the example for $O(\log n)$ – RichardOD Feb 23 '10 at 13:23

5 It's important to realize that its log base 2 (not base 10). This is because at each step in an algorithm, you remove half of your remaining choices. In computer science we almost always deal with log base 2 because we can ignore constants. However there are some exceptions (i.e. Quad Tree run times are log base 4) – Ethan May 27 '13 at 1:33

7 @Ethan: It doesn't matter which base you are in, since base conversion is just a constant multiplication. The formula is $\log_b(x) = \log_d(x) / \log_d(b)$. $\log_d(b)$ will just be a constant. – mindvirus May 27 '13 at 1:46

29 Answers

I cannot understand how to identify a function with a log time.

The most common attributes of logarithmic running-time function are that:

- the choice of the next element on which to perform some action is one of several possibilities, and
- only one will need to be chosen.

or

- the elements on which the action is performed are digits of n

This is why, for example, looking up people in a phone book is $O(\log n)$. You don't need to check every person in the phone book to find the right one; instead, you can simply divide-and-conquer by looking based on where their name is alphabetically, and in every section you only need to explore a subset of the each section before you eventually find someone's phone number.

Of course, a bigger phone book will still take you a longer time, but it won't grow as quickly as the proportional increase in the additional size.

We can expand the phone book example to compare other kinds of operations and *their* running time. We will assume our phone book has *businesses* (the "Yellow Pages") which have unique names and *people* (the "White Pages") which may not have unique names. A phone number is assigned to at most one person or business. We will also assume that it takes constant time to flip to a specific page.

Here are the running times of some operations we might perform on the phone book, from best to worst:

- **$O(1)$ (worst case):** Given the page that a business's name is on and the business name, find the phone number.
- **$O(1)$ (average case):** Given the page that a person's name is on and their name, find the phone number.
- **$O(\log n)$:** Given a person's name, find the phone number by picking a random point about halfway through the part of the book you haven't searched yet, then checking to see whether the person's name is at that point. Then repeat the process about halfway through the part of the book where the person's name lies. (This is a binary search for a person's name.)
- **$O(n)$:** Find all people whose phone numbers contain the digit "5".
- **$O(n)$:** Given a phone number, find the person or business with that number.
- **$O(n \log n)$:** There was a mix-up at the printer's office, and our phone book had all its pages inserted in a random order. Fix the ordering so that it's correct by looking at the first name on each page and then putting that page in the appropriate spot in a new, empty phone book.

For the below examples, we're now at the printer's office. Phone books are waiting to be mailed to each resident or business, and there's a sticker on each phone book identifying where it should be mailed to. Every person or business gets one phone book.

- **$O(n \log n)$:** We want to personalize the phone book, so we're going to find each person or business's name in their designated copy, then circle their name in the book and write a short thank-you note for their patronage.
- **$O(n^2)$:** A mistake occurred at the office, and every entry in each of the phone books has an extra "0" at the end of the phone number. Take some white-out and remove each zero.
- **$O(n \cdot n!)$:** We're ready to load the phonebooks onto the shipping dock. Unfortunately, the robot that was supposed to load the books has gone haywire: it's putting the books onto the truck in a random order! Even worse, it loads all the books onto the truck, then checks to see if they're in the right order, and if not, it unloads them and starts over. (This is the dreaded [bogo sort](#).)
- **$O(n^n)$:** You fix the robot so that it's loading things correctly. The next day, one of your co-workers plays a prank on you and wires the loading dock robot to the automated printing systems. Every time the robot goes to load an original book, the factory printer makes a duplicate run of all the phonebooks! Fortunately, the robot's bug-detection systems are sophisticated enough that the robot doesn't try printing even more copies when it encounters a duplicate book for loading, but it still has to load every original and duplicate book that's been printed.

edited Apr 4 at 2:34



aug

4,093 3 26 57

answered Feb 21 '10 at 20:14



John Feminella

190k 30 284 311


explanation! A win all-around. (Also, it looks like your answer was made before I was even a member on StackOverflow to begin with!) – [John Feminella](#) Feb 23 '10 at 0:40

6 "A mistake occurred at the office, and every entry in each of the phone books has an extra "0" at the end of the phone number. Take some white-out and remove each zero." <-- this is not order N squared. N is defined as the size of the input. The size of the input is the number of phone numbers, which is the number of numbers per book times the number of books. That's still a linear time operation. – [Billy O'Neal](#) Apr 10 '10 at 17:13

11 @Billy: In this example, N is the number of people in a single book. Because every person in the phone book also gets their own copy of the book, there are N identical phone books, each with N people in it, which is $O(N^2)$. – [John Feminella](#) Apr 10 '10 at 17:32


38 Isn't $O(1)$ the best case, rather than worst case as it is strangely highlighted as? – [Svip](#) May 26 '13 at 8:29

32 It took me $O(\log n)$ time to find an $O(\log n)$ definition which finally makes sense. +1 – [iAteABug_And_iLiked_it](#) Aug 30 '13 at 17:19



Love remote work?

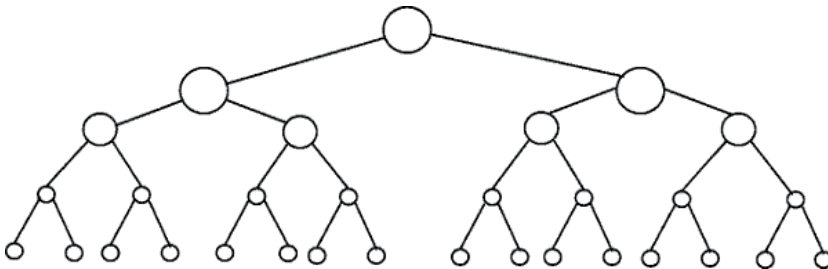
Find it on a new kind of career site


[Get started](#)

Many good answers have already been posted to this question, but I believe we really are missing an important one - namely, the illustrated answer.

What does it mean to say that the height of a complete binary tree is $O(\log n)$?

The following drawing depicts a binary tree. Notice how each level contains double the number of nodes compared to the level above (hence *binary*):



Binary search is an example with complexity $O(\log n)$. Let's say that the nodes in the bottom level of the tree in figure 1 represents items in some sorted collection. Binary search is a divide-and-conquer algorithm, and the drawing shows how we will need (at most) 4 comparisons to find the record we are searching for in this 16 item dataset.

Assume we had instead a dataset with 32 elements. Continue the drawing above to find that we will now need 5 comparisons to find what we are searching for, as the tree has only grown one level deeper when we multiplied the amount of data. As a result, the complexity of the algorithm can be described as a logarithmic order.

Plotting $\log(n)$ on a plain piece of paper, will result in a graph where the rise of the curve decelerates as n increases:



edited Apr 17 '16 at 11:06



sahil
2,236 1 5 32

answered Feb 21 '10 at 22:22



Jørn Schou-Rode
26.8k 13 64 108

36 "Notice how each level contains the double number of nodes compared to the level above (hence binary)" This is incorrect. What you're describing is a *balanced* binary tree. A binary tree just means each node has at most two children. – [Oenotria](#) May 27 '13 at 18:28

5 In fact, it's a very special balanced binary tree, called a complete binary tree. I've edited the answer but need someone to approve it. – [user21820](#) Dec 14 '13 at 3:44

4 A complete binary tree doesn't need to have the last level to be completely filled. I would say, a 'full binary tree' is more appropriate. – [Mr. AJ](#) Jul 29 '14 at 1:21

Your answer tries to respond more concretely to the original problem of the OP, so it is better than the current accepted answer (IMO), but it is still very incomplete: you just give a half example and 2 images... – [nbro](#) Aug 12 '15 at 14:04

This tree has 31 items in it, not 16. Why is it called a 16 item data set? Every node on it represents a number, otherwise it would be an inefficient binary tree :P – [Perry Monschau](#) Nov 23 '16 at 21:36

$O(\log N)$ basically means time goes up linearly while the N goes up exponentially. So if it takes 1 second to compute 10 elements, it will take 2 seconds to compute 100 elements, 3 seconds to compute 1000 elements, and so on.

It is $O(\log n)$ when we do divide and conquer type of algorithms e.g binary search. Another example is quick sort where each time we divide the array into two parts and each time it takes $O(N)$ time to find a pivot element. Hence it is $N O(\log N)$

edited Dec 29 '14 at 20:27

answered Feb 21 '10 at 20:18



[fastcodejava](#)

19.5k 17 97 145

3 concise and clear answer! loved it!! – [Shankar](#) Nov 17 '13 at 4:57

28 Three lines of wisdom that beats all other essay answers... :) Just in case somebody is missing it, in programming context, the base of log is 2 (not 10), so $O(\log n)$ scales like 1 sec for 10 elements, 2 sec for 20, 3 for 40 etc. – [nawfal](#) May 23 '14 at 10:32

2 yes, logarithmic function is its inverse to exponential function. $((\log x) \text{ base } a)$ is inverse of $(a \text{ power } x)$. Qualitative analysis of these functions with graphs would give more intuition. – [overexchange](#) Oct 16 '14 at 1:02

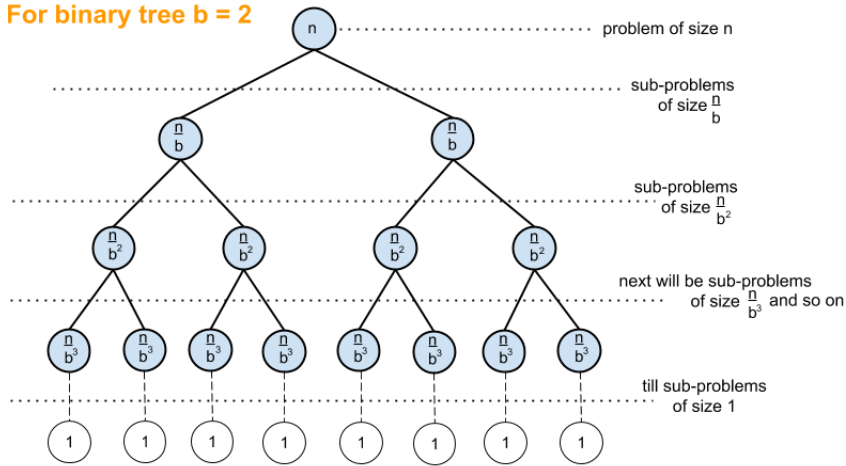
2 This answer is simple and clear. – [CarlLee](#) Mar 31 '15 at 0:58

2 This took me about 3 read-throughs to realize it wasn't wrong. *Time* goes up linearly, while the *number of elements* is exponential. This means **more elements during less time**. This is mentally taxing for those that visualize \log as the familiar log curve on a graph. – [Qix](#) Feb 15 at 9:30

The explanation below is using the case of a fully *balanced* binary tree to help you understand how we get logarithmic time complexity.

Binary tree is a case where a problem of size n is divided into sub-problem of size $n/2$ until we reach a problem of size 1:

For binary tree $b = 2$



The height of the above tree is answer to the following question: How many times we divide problem of size n by b until we get down to problem of size 1?

The other way of asking same question:

when $\frac{n}{b^x} = 1$ [in binary tree $b = 2$]

i.e. $n = b^x$ which is $\log_b n$ [by definition of logarithm]

And that's how you get $O(\log n)$ which is the amount of work that needs to be done on the above tree to reach a solution.

A common algorithm with $O(\log n)$ time complexity is Binary Search whose recursive relation is $T(n/2) + O(1)$ i.e. at every subsequent level of the tree you divide problem into half and do constant amount of additional work.

edited Feb 23 '16 at 15:31

answered Oct 26 '12 at 19:33



2cupsOfTech

3,193 2 21 40

2 newbie here. So could you say the tree height is the division rate by recursion to reach size $n=1$? – Cody Nov 12 '13 at 8:54

4 Brilliant explanation! – Olumide Sep 11 '15 at 18:31

5 This is definitely the best answer here – Yavar Jan 12 '16 at 3:11

3 Though there are a lot of good answers in this thread, this one is the best. Thanks! – Jonck van der Kogel Jun 1 '16 at 6:38

1 This one explained it perfectly! – Sina Madani Aug 6 '16 at 0:56

If you had a function that takes:

1 millisecond to complete if you have 2 elements.
2 milliseconds to complete if you have 4 elements.
3 milliseconds to complete if you have 8 elements.
4 milliseconds to complete if you have 16 elements.
...
 n milliseconds to complete if you have 2^{*n} elements.

Then it takes $\log_2(n)$ time. The Big O notation, loosely speaking, means that the relationship only needs to be true for large n , and that constant factors and smaller terms can be ignored.

answered Feb 21 '10 at 20:11



Mark Byers

481k 97 1159 1232

10 The most simple and useful answer – Veehmot Aug 19 '12 at 0:24

This was a fantastic answer. Thank you! – Omar N Nov 10 '15 at 0:51

thank you for good answer. – Abc Aug 8 '16 at 5:14

Logarithmic running time ($O(\log n)$) essentially means that the running time grows in proportion to the *logarithm* of the input size - as an example, if 10 items takes at most some amount of time x , and 100 items takes at most, say, $2x$, and 10,000 items takes at most $4x$, then it's looking like an $O(\log n)$ time complexity.

edited Feb 21 '10 at 20:27

answered Feb 21 '10 at 20:10



Anon.

36.4k 4 58 80

51 log2 or log10 is irrelevant. They only differ by a scale factor, which makes them of the same order, i.e. they still grow at the same rate. – Noldorin Feb 21 '10 at 20:16

14 The fun thing about logarithms is that when comparing relative heights, the exact base you use doesn't matter. $\log_{10,000} / \log_{100}$ is 2 regardless of what base you use. – Anon. Feb 21 '10 at 20:18

11 To be nitpicky, $O(\lg n)$ means that the runtime is *at most* proportional to $\lg n$. What you describe is $\Theta(\lg n)$. – anon Feb 21 '10 at 20:22

1 @rgrig: That is true. I've edited in a few "at mosts" to indicate the upper-bound nature of big-O. – Anon. Feb 21 '10 at 20:27

1 @rgrig he described both O and theta: $\Theta(\lg n)$ implies $O(\lg n)$ – klochner Feb 21 '10 at 20:43

You can think of $O(\log N)$ intuitively by saying the time is proportional to the number of digits in N .

If an operation performs constant time work on each digit or bit of an input, the whole operation will take time proportional to the number of digits or bits in the input, not the magnitude of the input; thus, $O(\log N)$ rather than $O(N)$.

If an operation makes a series of constant time decisions each of which halves (reduces by a factor of 3, 4, 5...) the size of the input to be considered, the whole will take time proportional to \log base 2 (base 3, base 4, base 5...) of the size N of the input, rather than being $O(N)$.

And so on.

answered Feb 21 '10 at 20:13



moonshadow

45.5k 6 61 108

5 Accurate enough and more easily grasped than most explanations, I reckon. – T. Feb 21 '10 at 20:35

it's an explanation of $\log_{10} N$, is it? – LiuYan 刘研 Apr 14 '11 at 8:45

@LiuYan刘研 they didn't say what base the number of digits was in. In any case though, $\log_2(n) = \log_{10}(n)/\log_{10}(2)$ and $1/\log_{10}(2)$ is hence a constant multiplier, with the same principle applying to all other bases. This shows two things. Firstly that moonshadow's principle applies whatever the base (though the lower the base, the fewer "jags" in the estimate) and also that $O(\log n)$ is $O(\lg n)$ no matter what base the calculation that led you to that conclusion. – Jon Hanna Sep 2 '12 at 22:04

@JonHanna, i got it, thanks for the explanation. – LiuYan 刘研 Sep 3 '12 at 1:50

Overview

Others have given good diagram examples, such as the tree diagrams. I did not see any simple code examples. So in addition to my explanation, I'll provide some algorithms with simple print statements to illustrate the complexity of different algorithm categories.

First, you'll want to have a general idea of Logarithm, which you can get from <https://en.wikipedia.org/wiki/Logarithm>. Natural science use e and the natural log. Engineering disciplines will use \log_{10} (log base 10) and computer scientists will use \log_2 (log base 2) a lot, since computers are binary based. Sometimes you'll see abbreviations of natural log as $\ln()$, engineers normally leave the $_{10}$ off and just use $\log()$ and \log_2 is abbreviated as $\lg()$. All of the types of logarithms grow in a similar fashion, that is why they share the same category of $\log(n)$.

When you look at the code examples below, I recommend looking at $O(1)$, then $O(n)$, then $O(n^2)$. After you are good with those, then look at the others. I've included clean examples as well as variations to demonstrate how subtle changes can still result in the same categorization.

You can think of $O(1)$, $O(n)$, $O(\log n)$, etc as classes or categories of growth. Some categories will take more time to do than others. These categories help give us a way of ordering the algorithm performance. Some grown faster as the input n grows. The following table demonstrates said growth numerically. In the table below think of $\log(n)$ as the ceiling of \log_2 .

Input Size (n)	O(1)	O(log(n))	O(n)	O(n*log(n))	O(n^2)
1	1	1	1	1	1
4	1	2	4	8	16
16	1	4	16	64	256
1024	1	10	1024	10240	1048576

Simple Code Examples Of Various Big O Categories:

O(1) - Constant Time Examples:

- Algorithm 1:

Algorithm 1 prints hello once and it doesn't depend on n, so it will always run in constant time, so it is $O(1)$.

```
print "hello";
```

- Algorithm 2:

Algorithm 2 prints hello 3 times, however it does not depend on an input size. Even as n grows, this algorithm will always only print hello 3 times. That being said 3, is a constant, so this algorithm is also $O(1)$.

```
print "hello";
print "hello";
print "hello";
```

O(log(n)) - Logarithmic Examples:

- Algorithm 3 - This acts like "log₂"

Algorithm 3 demonstrates an algorithm that runs in $\log_2(n)$. Notice the post operation of the for loop multiplies the current value of i by 2, so i goes from 1 to 2 to 4 to 8 to 16 to 32 ...

```
for(int i = 1; i <= n; i = i * 2)
    print "hello";
```

- Algorithm 4 - This acts like "log₃"

Algorithm 4 demonstrates \log_3 . Notice i goes from 1 to 3 to 9 to 27...

```
for(int i = 1; i <= n; i = i * 3)
    print "hello";
```

- Algorithm 5 - This acts like "log_{1.02}"

Algorithm 5 is important, as it helps show that as long as the number is greater than 1 and the result is repeatedly multiplied against itself, that you are looking at a logarithmic algorithm.

```
for(double i = 1; i < n; i = i * 1.02)
    print "hello";
```

O(n) - Linear Time Examples:

- Algorithm 6

This algorithm is simple, which prints hello n times.

```
for(int i = 0; i < n; i++)
    print "hello";
```

- Algorithm 7

This algorithm shows a variation, where it will print hello $n/2$ times. $n/2 = 1/2 * n$. We ignore the $1/2$ constant and see that this algorithm is $O(n)$.

```
for(int i = 0; i < n; i = i + 2)
    print "hello";
```

O(n*log(n)) - nlog(n) Examples:

- Algorithm 8

Think of this as a combination of $O(\log(n))$ and $O(n)$. The nesting of the for loops help us obtain the $O(n*\log(n))$

```
for(int i = 0; i < n; i++)
    for(int j = 1; j < n; j = j * 2)
        print "hello";
```

- Algorithm 9

Algorithm 9 is like algorithm 8, but each of the loops has allowed variations, which still result in the final result being $O(n \log(n))$

```
for(int i = 0; i < n; i = i + 2)
  for(int j = 1; j < n; j = j * 3)
    print "hello";
```

$O(n^2)$ - n squared Examples:

- Algorithm 10

$O(n^2)$ is obtained easily by nesting standard for loops.

```
for(int i = 0; i < n; i++)
  for(int j = 0; j < n; j++)
    print "hello";
```

- Algorithm 11

Like algorithm 10, but with some variations.

```
for(int i = 0; i < n; i++)
  for(int j = 0; j < n; j = j + 2)
    print "hello";
```

$O(n^3)$ - n cubed Examples:

- Algorithm 12

This is like algorithm 10, but with 3 loops instead of 2.

```
for(int i = 0; i < n; i++)
  for(int j = 0; j < n; j++)
    for(int k = 0; k < n; k++)
      print "hello";
```

- Algorithm 13

Like algorithm 12, but with some variations that still yield $O(n^3)$.

```
for(int i = 0; i < n; i++)
  for(int j = 0; j < n + 5; j = j + 2)
    for(int k = 0; k < n; k = k + 3)
      print "hello";
```

Summary

The above give several straight forward examples, and variations to help demonstrate what subtle changes can be introduced that really don't change the analysis. Hopefully it gives you enough insight.

edited Apr 27 '16 at 14:35

answered Apr 26 '16 at 22:50



James Oravec

6,711 10 47 91

2 Awesome. The best explanation for me I ever seen. It'd be nicer if $O(n^2)$ is noted as a combination of $O(n)$ and $O(n)$, so $O(n) * O(n) = O(n * n) = O(n^2)$. It feels like a bit of jumping without this equation. This is repetition of prior explanation, but I think this repetition can provide more confidence to readers for understanding. — Eonil Jul 2 '16 at 2:19

The best way I've always had to mentally visualize an algorithm that runs in $O(\log n)$ is as follows:

If you increase the problem size by a multiplicative amount (i.e. multiply its size by 10), the work is only increased by an additive amount.

Applying this to your binary tree question so you have a good application: if you double the number of nodes in a binary tree, the height only increases by 1 (an additive amount). If you double it again, it still only increased by 1. (Obviously I'm assuming it stays balanced and such). That way, instead of doubling your work when the problem size is multiplied, you're only doing very slightly more work. That's why $O(\log n)$ algorithms are awesome.

answered Feb 22 '10 at 19:51



DivineWolfwood

1,392 8 19

this is the best answer ! — Sujal Mandal Aug 27 '16 at 18:27

The logarithm

Ok let's try and fully understand what a logarithm actually is.

Imagine we have a rope and we have tied it to a horse. If the rope is directly tied to the horse, the force the horse would need to pull away (say, from a man) is directly 1.

Now imagine the rope is looped round a pole. The horse to get away will now have to pull many times harder. The amount of times will depend on the roughness of the rope and the size of the pole, but let's assume it will multiply one's strength by 10 (when the rope makes a complete turn).

Now if the rope is looped once, the horse will need to pull 10 times harder. If the human decides to make it really difficult for the horse, he may loop the rope again round a pole, increasing it's strength by an additional 10 times. A third loop will again increase the strength by a further 10 times.



We can see that for each loop, the value increases by 10. The number of turns required to get any number is called the logarithm of the number i.e. we need 3 posts to multiply your strength by 1000 times, 6 posts to multiply your strength by 1,000,000.

3 is the logarithm of 1,000, and 6 is the logarithm of 1,000,000 (base 10).

So what does $O(\log n)$ actually mean?

In our example above, our 'growth rate' is $O(\log n)$. For every additional loop, the force our rope can handle is 10 times more:

Turns	Max Force
0	1
1	10
2	100
3	1000
4	10000
n	10^n

Now the example above did use base 10, but fortunately the base of the log is insignificant when we talk about big o notation.

Now let's imagine you are trying to guess a number between 1-100.

```
Your Friend: Guess my number between 1-100!
Your Guess: 50
Your Friend: Lower!
Your Guess: 25
Your Friend: Lower!
Your Guess: 13
Your Friend: Higher!
Your Guess: 19
Your Friend: Higher!
Your Friend: 22
Your Guess: Lower!
Your Guess: 20
Your Friend: Higher!
Your Guess: 21
Your Friend: YOU GOT IT!
```

Now it took you 7 guesses to get this right. But what is the relationship here? What is the most amount of items that you can guess from each additional guess?

Guesses	Items
1	2
2	4
3	8
4	16
5	32
6	64
7	128
10	1024

Using the graph, we can see that if we use a binary search to guess a number between 1-100 it will take us **at most** 7 attempts. If we had 128 numbers, we could also guess the number in 7 attempts but 129 numbers will take us **at most** 8 attempts (in relations to logarithms, here we would need 7 guesses for a 128 value range, 10 guesses for a 1024 value range. 7 is the logarithm of 128, 10 is the logarithm of 1024 (base 2)).

Notice that I have bolded 'at most'. Big o notation always refers to the worse case. If you're lucky, you could guess the number in one attempt and so the best case is $O(1)$, but that's another story.

We can see that for every guess our data set is shrinking. A good rule of thumb to identify if an algorithm has a logarithmic time is to see if the data set shrinks by a certain order after each iteration

What about $O(n \log n)$?

You will eventually come across a linearithmic time $O(n \log(n))$ algorithm. The rule of thumb above applies again, but this time the logarithmic function has to run n times e.g. reducing the size of a list n times, which occurs in algorithms like a mergesort.

You can easily identify if the algorithmic time is $n \log n$. Look for an outer loop which iterates through a list ($O(n)$). Then look to see if there is an inner loop. If the inner loop is **cutting/reducing** the data set on each iteration, that loop is ($O(\log n)$), and so the overall algorithm is $= O(n \log n)$.

Disclaimer: The rope-logarithm example was grabbed from the excellent [Mathematician's Delight](#) book by W.Sawyer.

edited Oct 14 '16 at 11:46

answered Oct 8 '16 at 14:27



[gudthing](#)

2,915 2 19 35

1 great... Thanks – [blackDelta-Δ](#) Oct 14 '16 at 8:17

Superb !! @gudthing – [Rajasuba Subramanian](#) Feb 27 at 17:42

What's $\log_b(n)$?

It is the number of times you can cut a log of length n repeatedly into b equal parts before reaching a section of size 1.

edited Oct 26 '16 at 17:58

answered Mar 19 '10 at 19:28



[James Oravec](#)

6,711 10 47 91



[Chad Brewbaker](#)

1,520 12 21

Divide and conquer algorithms usually have a $\log n$ component to the running time. This comes from the repeated halving of the input.

In the case of binary search, every iteration you throw away half of the input. It should be noted that in Big-O notation, log is log base 2.

Edit: As noted, the log base doesn't matter, but when deriving the Big-O performance of an algorithm, the log factor will come from halving, hence why I think of it as base 2.

edited Feb 21 '10 at 20:27

answered Feb 21 '10 at 20:11



[David Kanarek](#)

11.1k 4 37 59

2 Why is it log base 2? In randomized quicksort for example, I don't think it is base 2. As far as I know, the base doesn't matter, as $\log_{base a}(n) = \log_2(n) / \log_2(a)$, so every logarithm is different from another by a constant, and constants are ignored in big-o notation. In fact, writing the base of a log in big-o notation is a mistake in my opinion, as you are writing a constant. – [IVlad](#) Feb 21 '10 at 20:14

1 Re "log is log base 2": [stackoverflow.com/questions/1569702/is-big-ologn-log-base-e/...](#) – [user200783](#) Feb 21 '10 at 20:15

Very true that it can be converted to any base and it does not matter, but if you are trying to derive the Big-O performance and you see constant halving, it helps to understand that you won't see log base 10 reflected in the code. – [David Kanarek](#) Feb 21 '10 at 20:25

An aside: In things such as B-trees, where nodes have a fan-out of more than 2 (i.e. "wider" than a binary tree), you'll still see $O(\log n)$ growth, because it's still divide-and-conquer, but the base of the log will be related to the fan-out. – [Roger Lipscombe](#) Feb 22 '10 at 19:30

But what exactly is $O(\log n)$? For example, what does it mean to say that the height of a complete binary tree is $O(\log n)$?

I would rephrase this as 'height of a complete binary tree is $\log n$ '. Figuring the height of a complete binary tree would be $O(\log n)$, if you were traversing down step by step.

I cannot understand how to identify a function with a logarithmic time.

Logarithm is essentially the inverse of exponentiation. So, if each 'step' of your function is eliminating a **factor** of elements from the original item set, that is a logarithmic time algorithm.

For the tree example, you can easily see that stepping down a level of nodes cuts down an exponential number of elements as you continue traversing. The popular example of looking through a name-sorted phone book is essentially equivalent to traversing down a binary search tree (middle page is the root element, and you can deduce at each step whether to go left or right).

answered May 26 '13 at 8:35



[user2421873](#)

165 1 2

3 +1 for mentioning "Logarithm is essentially the inverse of exponentiation". – [talonx](#) Nov 16 '13 at 15:13

$O(\log n)$ refers to a function (or algorithm, or step in an algorithm) working in an amount of time proportional to the logarithm (usually base 2 in most cases, but not always, and in any event this is insignificant by big-O notation*) of the size of the input.

The logarithmic function is the inverse of the exponential function. Put another way, if your input grows exponentially (rather than linearly, as you would normally consider it), your function grows linearly.

$O(\log n)$ running times are very common in any sort of divide-and-conquer application, because you are (ideally) cutting the work in half every time. If in each of the division or conquer steps, you are doing constant time work (or work that is not constant-time, but with time growing more slowly than $O(\log n)$), then your entire function is $O(\log n)$. It's fairly common to have each step require linear time on the input instead; this will amount to a total time complexity of $O(n \log n)$.

The running time complexity of binary search is an example of $O(\log n)$. This is because in binary search, you are always ignoring half of your input in each later step by dividing the array in half and only focusing on one half with each step. Each step is constant-time, because in binary search you only need to compare one element with your key in order to figure out what to do next irregardless of how big the array you are considering is at any point. So you do approximately $\log(n)/\log(2)$ steps.

The running time complexity of merge sort is an example of $O(n \log n)$. This is because you are dividing the array in half with each step, resulting in a total of approximately $\log(n)/\log(2)$ steps. However, in each step you need to perform merge operations on all elements (whether it's one merge operation on two sublists of $n/2$ elements, or two merge operations on four sublists of $n/4$ elements, is irrelevant because it adds to having to do this for n elements in each step). Thus, the total complexity is $O(n \log n)$.

*Remember that big-O notation, [by definition](#), constants don't matter. Also by the [change of base rule](#) for logarithms, the only difference between logarithms of different bases is a constant factor.

answered Feb 21 '10 at 20:19



[Platinum Azure](#)

30k 2 75 113

The final * note solved my confusion about logarithms being based on 2 or 10 :) Thanks a lot. – [yahya](#) Sep 18 '15 at 9:50

Simply put: At each step of your algorithm you can cut the work in half. (Asymptotically equivalent to third, fourth, ...)

answered Feb 22 '10 at 16:41



[Brian R. Bondy](#)

212k 87 495 577

This answer is very imprecise. First of all, you can think of cutting the work in half only in the case of the logarithm in base 2. It's really incredible how this answer (and most of the answers to the original question) received so many up-votes. "(Asymptotically equivalent to third, fourth, ...)"? Why answering a question if you don't have time? – [nbro](#) Feb 20 '16 at 16:05

If you plot a logarithmic function on a graphical calculator or something similar, you'll see that it rises really slowly -- even more slowly than a linear function.

This is why algorithms with a logarithmic time complexity are highly sought after: even for really big n (let's say $n = 10^8$, for example), they perform more than acceptably.

edited Aug 26 '10 at 9:48

answered Feb 21 '10 at 20:11



[Hadewijch Debaillie](#)

119 6

These 2 cases will take $O(\log n)$ time

```
case 1: f(int n) {
    int i;
    for (i = 1; i < n; i=i*2)
        printf("%d", i);
}
```

```
case 2 : f(int n) {
    int i;
    for (i = n; i>=1; i=i/2)
        printf("%d", i);
}
```

edited Dec 26 '14 at 12:58

answered Jul 5 '13 at 3:43



[Ravi Bisla](#)

185 2 7

I'm sure I'm missing something, but wouldn't i always be zero and the loops run forever in both of those cases, since $0*2=0$ and $0/2=0$? – [dj_segfault](#) Sep 29 '13 at 19:13

2 @dj_segfault, that was my mistake. I think now it does make sense... – [Ravi Bisla](#) Sep 30 '13 at 18:29

It simply means that the time needed for this task grows with $\log(n)$ (example : 2s for $n = 10$, 4s for $n = 100$, ...). Read the Wikipedia articles on [Binary Search Algorithm](#) and [Big O Notation](#) for more precisions.

answered Feb 21 '10 at 20:10



[Valentin Rocher](#)

9,581 31 54

But what exactly is $O(\log n)$

What it means precisely is "as n tends towards infinity, the time tends towards $a \cdot \log(n)$ where a is a constant scaling factor".

Or actually, it doesn't quite mean that; more likely it means something like "time divided by $a \cdot \log(n)$ tends towards 1".

"Tends towards" has the usual mathematical meaning from 'analysis': for example, that "if you pick *any* arbitrarily small non-zero constant k , then I can find a corresponding value x such that $((\text{time}/(a \cdot \log(n))) - 1)$ is less than k for all values of n greater than x ".

In lay terms, it means that the equation for time may have some other components: e.g. it may have some constant startup time; but these other components pale towards insignificance for large values of n , and the $a \cdot \log(n)$ is the dominating term for large n .

Note that if the equation were, for example ...

$\text{time}(n) = a + b \log(n) + cn + dn^2$

... then this would be $O(n^2)$ because, no matter what the values of the constants a , b , c , and non-zero d , the $d \cdot n^2$ term would always dominate over the others for any sufficiently large

value of n .

That's what bit O notation means: it means "what is the order of dominant term for any sufficiently large n ".

edited Feb 21 '10 at 20:44

answered Feb 21 '10 at 20:32



ChrisW

43.2k 6 76 165

That is wrong. en.wikipedia.org/wiki/... – Michael Graczyk Jul 16 '12 at 11:29

I can add something interesting, that I read in book by Kormen and etc. a long time ago. Now, imagine a problem, where we have to find a solution in a problem space. This problem space should be finite.

Now, if you can prove, that at every iteration of your algorithm you cut off a fraction of this space, that is no less than some limit, this means that your algorithm is running in $O(\log N)$ time.

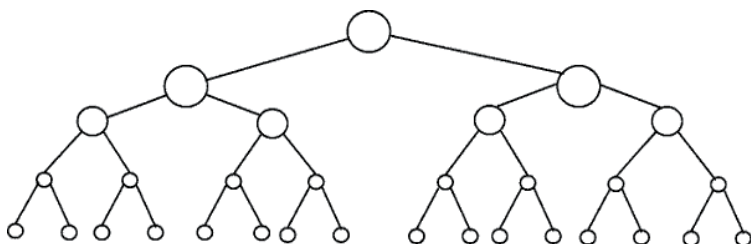
I should point out, that we are talking here about a relative fraction limit, not the absolute one. The binary search is a classical example. At each step we throw away $1/2$ of the problem space. But binary search is not the only such example. Suppose, you proved somehow, that at each step you throw away at least $1/128$ of problem space. That means, your program is still running at $O(\log N)$ time, although significantly slower than the binary search. This is a very good hint in analyzing of recursive algorithms. It often can be proved that at each step the recursion will not use several variants, and this leads to the cutoff of some fraction in problem space.

answered Feb 4 '14 at 12:52



SPIRiT_1984

1,263 1 16 28



$\log x$ to base $b = y$ is the inverse of $b^y = x$

If you have an M -ary tree of depth d and size n , then:

- traversing the whole tree $\sim O(M^d) = O(n)$
- Walking a single path in the tree $\sim O(d) = O(\log n \text{ to base } M)$

answered May 28 '13 at 7:07



Khaled.K

4,412 1 21 38

$O(\log n)$ is a bit misleading, more precisely it's $O(\log_2 n)$, i.e. (logarithm with base 2).

The height of a balanced binary tree is $O(\log_2 n)$, since every node has two (note the "two" as in $\log_2 n$) child nodes. So, a tree with n nodes has a height of $\log_2 n$.

Another example is binary search, which has a running time of $O(\log_2 n)$ because at every step you divide the search space by 2.

edited Feb 20 '16 at 15:57

answered Feb 21 '10 at 20:12



nbro

4,316 5 20 57



stmax

4,143 2 17 31

4 $O(\log n)$ is the same order as $O(\text{ld } n)$ or $O(\text{LN } n)$. They are proportional. I understand that for learning purposes it's easier to use ld . – helios Feb 21 '10 at 20:14

4 "more precisely it's $O(\text{ld } n)$ " - No, it isn't: all logs are the same order (each differing from the others only by some constant scaling factor, which is ignored/ignorable). – ChrisW Feb 21 '10 at 20:23

you're right chris, very bad wording. should have said it as helios did. it helps for learning/understanding but finally all logs are the same order. – [stmax](#) Feb 23 '10 at 17:08

I can give an example for a for loop and maybe once grasped the concept maybe it will be simpler to understand in different contexts.

That means that in the loop the step grows exponentially. E.g.

```
for (i=1; i<=n; i=i*2) {}
```

The complexity in O-notation of this program is $O(\log(n))$. Let's try to loop through it by hand (n being somewhere between 512 and 1023 (excluding 1024):

step:	1	2	3	4	5	6	7	8	9	10
i:	1	2	4	8	16	32	64	128	256	512

Although n is somewhere between 512 and 1023, only 10 iterations take place. This is because the step in the loop grows exponentially and thus takes only 10 iterations to reach the termination.

The logarithm of x (to the base of a) is the reverse function of a^x .

It is like saying that logarithm is the inverse of exponential.

Now try to see it that way, if exponential grows very fast then logarithm grows (inversely) very slow.

The difference between $O(n)$ and $O(\log(n))$ is huge, similar to the difference between $O(n)$ and $O(a^n)$ (a being a constant).

answered May 16 '15 at 4:27



[Elyasin](#)

5,818 2 19 42

In information technology it means that:

$f(n)=O(g(n))$ If there is suitable constant C and N_0 independent on N,
such that
for all $N>N_0$ " $C \cdot g(n) > f(n) > 0$ " is true.

Ant it seems that this notation was mostly have taken from mathematics.

In this article there is a quote: [D.E. Knuth, "BIG OMICRON AND BIG OMEGA AND BIG THETA", 1976](#):

On the basis of the issues discussed here, I propose that members of SIGACT, and editors of computer science and mathematics journals, adopt notations as defined above, unless a **better alternative can be found reasonably soon**.

Today is 2016, but we use it still today.

In mathematical analysis it means that:

$\lim (f(n)/g(n))=\text{Constant}$; where n goes to +infinity

But even in mathematical analysis sometimes this symbol was used in meaning " $C \cdot g(n) > f(n) > 0$ ".

As I know from university the symbol was introduced by German mathematician Landau (1877-1938)

edited Aug 28 '16 at 16:12

answered Apr 2 '14 at 11:37



[bruziuz](#)

1,321 10 23

The complete binary example is $O(\ln n)$ because the search looks like this:

1 2 3 4 5 6 7 8 9 10 11 12

Searching for 4 yields 3 hits: 6, 3 then 4. And $\log_2 12 = 3$, which is a good approximate to how many hits where needed.

answered Feb 21 '10 at 20:11



Amirshk

6,119 1 18 60

thanks for the example. It clearly says how our algorithm can use logarithmic time in divide and conquer method. – [Abc](#) Aug 8 '16 at 5:06

So if its a loop of $n/2$ its always $\log(n)$? – [Gil Beyruth](#) Dec 10 '16 at 20:40

If you are looking for a intuition based answer I would like to put up two interpretations for you.

1. Imagine a very high hill with a very broad base as well. To reach the top of the hill there are two ways: one is a dedicated pathway going spirally around the hill reaching at the top, the other: small terrace like carvings cut out to provide a staircase. Now if the first way is reaching in linear time $O(n)$, the second one is $O(\log n)$.
2. Imagine an algorithm, which accepts an integer, n as input and completes in time proportional to n then it is $O(n)$ or $\theta(n)$ but if it runs in time proportion to the number of digits or the number of bits in the binary representation on number then the algorithm runs in $O(\log n)$ or $\theta(\log n)$ time.

edited Feb 21 '16 at 9:46

answered May 26 '13 at 16:56



mickeymoon

1,848 3 18 29

please edit. has " $O(n)$ or $\theta(n)$ " in both scenarios...? Also, I've heard this a lot, the size vs the # digits. Are we saying size == 128 for $n=10000000$ and digits == 8 for $n=10000000$? Please elucidate. – [Cody](#) Nov 12 '13 at 10:01

Note sure if your analogy is good. – [nbro](#) Feb 20 '16 at 16:40

I would like to add that the height of the tree is the length of the longest path from the root to a leaf, and that the height of a node is the length of the longest path from that node to a leaf. The path means the number of nodes we encounter while traversing the tree between two nodes. In order to achieve $O(\log n)$ time complexity, the tree should be balanced, meaning that the difference of the height between the children of any node should be less than or equal to 1. Therefore, trees do not always guarantee a time complexity $O(\log n)$, unless they are balanced. Actually in some cases, the time complexity of searching in a tree can be $O(n)$ in the worst case scenario.

You can take a look at the balance trees such as AVL tree. This one works on balancing the tree while inserting data in order to keep a time complexity of $(\log n)$ while searching in the tree.

edited Dec 2 '13 at 0:10

answered Dec 1 '13 at 23:52



Traveling Salesman

708 5 19 46

Algorithms in the Divide and Conquer paradigm are of complexity $O(\log n)$. One example here, calculate your own power function,

```
int power(int x, unsigned int y)
{
    int temp;
    if( y == 0)
        return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else
        return x*temp*temp;
}
```

from <http://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/>

answered Jun 27 '15 at 8:20



octoback

10.4k 21 81 140

Actually, if you have a list of n elements, and create a binary tree from that list (like in the divide and conquer algorithm), you will keep dividing by 2 until you reach lists of size 1 (the leaves).

At the first step, you divide by 2. You then have 2 lists (2^1), you divide each by 2, so you have 4 lists (2^2), you divide again, you have 8 lists (2^3) and so on until your list size is 1

That gives you the equation :

$$n/(2^{\text{steps}})=1 \iff n=2^{\text{steps}} \iff \lg(n)=\text{steps}$$

(you take the lg of each side, lg being the log base 2)

answered Nov 25 '16 at 18:50



[Dinaiz](#)

1,060 1 13 26

Until some malware begins to insert a new list with x length at two levels before the leaves nodes. Then it will seem to be an infinite loop... – [Francis Cugler](#) Feb 19 at 5:46

I didn't get your comment. Is my explanation wrong ? – [Dinaiz](#) Feb 28 at 11:19

I was only making a hypothetical joke. I wasn't really meaning anything by it. – [Francis Cugler](#) Mar 26 at 1:39

protected by [Andreas Grech](#) Nov 25 '13 at 12:06

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 [reputation](#) on this site (the [association bonus](#) does not count).

Would you like to answer one of these [unanswered questions](#) instead?