Khan Academy

Functions in asymptotic notation

Google Classroom Facebook Twitter Email

When we use asymptotic notation to express the rate of growth of an algorithm's running time in terms of the input size n, it's good to bear a few things in mind.

Let's start with something easy. Suppose that an algorithm took a constant amount of time, regardless of the input size. For example, if you were given an array that is already sorted into increasing order and you had to find the minimum element, it would take constant time, since the minimum element must be at index 0. Since we like to use a function of n in asymptotic notation, you could say that this algorithm runs in $\Theta(n^0)$ time. Why? Because $n^0 = 1$, and the algorithm's running time is within some constant factor of 1. In practice, we don't write $\Theta(n^0)$, however; we write $\Theta(1)$.

Now suppose an algorithm took $\Theta(\log_{10} n)$ time. You could also say that it took $\Theta(\log_2 n)$ time. Whenever the base of the logarithm is a constant, it doesn't matter what base we use in asymptotic notation. Why not? Because there's a mathematical formula that says

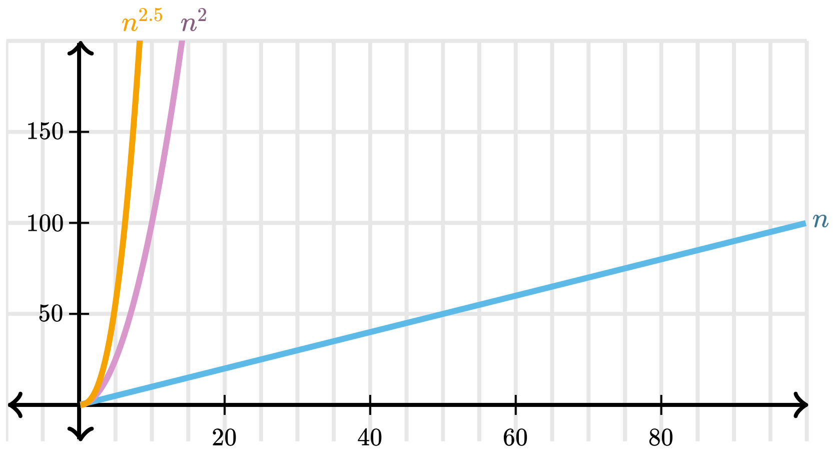
$$\log_a n = \frac{\log_b n}{\log_b a}$$

for all positive numbers a, b, and n. Therefore, if a and b are constants, then $\log_a n$ and $\log_b n$ differ only by a factor of $\log_b a$, and that's a constant factor which we can ignore in asymptotic notation.

Therefore, we can say that the worst-case running time of binary search is $\Theta(\log_a n)$ for any positive constant a. Why? The number of guesses is at most $\log_2 n + 1$, generating and testing each guess takes constant time, and setting up and returning take constant time. However, as a matter of practice, we often write that binary search takes $\Theta(\log_2 n)$ time because computer scientists like to think in powers of 2.

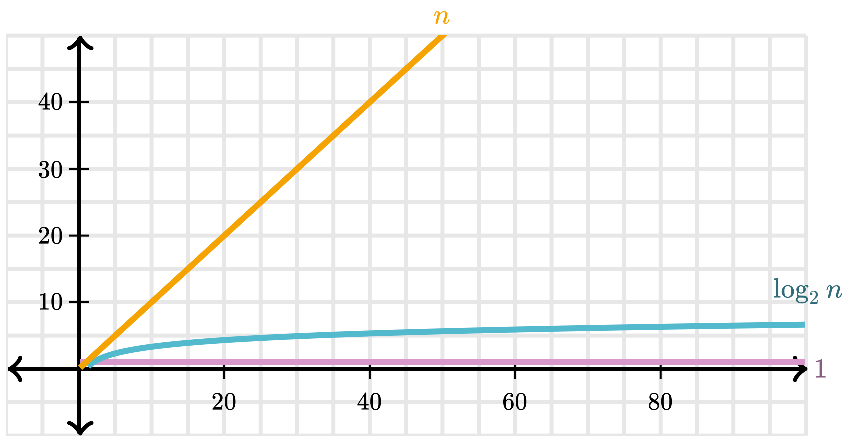
There is an order to the functions that we often see when we analyze algorithms using asymptotic notation. If a and b are constants and $a < b$, then a running time of $\Theta(n^a)$ grows more slowly than a running time of $\Theta(n^b)$. For example, a running time of $\Theta(n)$, which is $\Theta(n^1)$, grows more slowly than a running time of $\Theta(n^2)$. The exponents don't have to be integers, either. For example, a running time of $\Theta(n^2)$ grows more slowly than a running time of $\Theta(n^2\sqrt{n})$, which is $\Theta(n^{2.5})$.

The following graph compares the growth of n,n^2, and $n^{2.5}$:



Logarithms grow more slowly than polynomials. That is, $\Theta(\log_2 n)$ grows more slowly than $\Theta(n^a)$ for *any* positive constant a. But since the value of $\log_2 n$ increases as n increases, $\Theta(\log_2 n)$ grows faster than $\Theta(1)$.

The following graph compares the growth of 1, n, and $\log_2 n$:



Here's a list of functions in asymptotic notation that we often encounter when analyzing algorithms, ordered by slowest to fastest growing:

1. $\Theta(1)$
2. $\Theta(\log_2 n)$
3. $\Theta(n)$
4. $\Theta(n \log_2 n)$
5. $\Theta(n^2)$
6. $\Theta(n^2 \log_2 n)$
7. $\Theta(n^3)$
8. $\Theta(2^n)$
9. $\Theta(n!)$

Note that an exponential function a^n, where $a > 1$, grows faster than any polynomial function n^b, where b is any constant.

The list above is not exhaustive, there are many functions with running times not listed there. You'll hopefully run into a few of those in your computer science journey.

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

Sort by: Top Voted

Questions Tips & Thanks

Want to join the conversation?

You need at least 5000 energy points to get started.

Vijayenthiran Subramaniam 5 years ago

more

1. What does the prof meant by saying "The number of guesses is at most lgn+1" ?

2. Can some one explain with example for this statement:
Note that an exponential function a
n , where a>1, grows faster than any polynomial function n
b , where b is any constant.

1 comment (21 votes) Flag more

Cameron 5 years ago more

1)
A binary search performed on n items requires at most "log_2(n)+1" guesses. You might get lucky and find the right item before that, but if you follow a binary search algorithm you won't exceed log_2(n)+1 guesses before finding the right item.
e.g. If we have 8 items in a sorted listed the binary search algorithm will find the answer in: log_2(8)+1= 3+1 = 4 guesses or less

2)
e.g. 3^n grows faster than n^5

```
n=1   3^n=3       n^5=1
n=2   3^n=9       n^5=32
n=3   3^n=27      n^5=243
n=4   3^n=81      n^5=1024
n=5   3^n=243     n^5=3125
n=6   3^n=729     n^5=7776
n=7   3^n=2187    n^5=16807
n=8   3^n=6561    n^5=32768
n=9   3^n=19683   n^5=59049
n=10  3^n=59049   n^5=100000
n=11  3^n=177147  n^5=161051
n=12  3^n=531441  n^5=248832
```

After n=11, 3^n is larger than n^5, and the difference between them will become larger and larger as n increases

2 comments (97 votes) Flag more

See 6 more replies

Chase Eaton 5 years ago

more

It doesn't mention how the base on a log affects its growth. What does changing the base of a log do to its growth rate? The part about the change-of-base formula says that all logs with constant bases are equal but on the next quiz it is seen that they are not. Why is that so?

(8 votes) Flag more

Cameron 5 years ago

more

Functions can be in the same complexity class and have different growth rates

e.g.
'1000 * n' clearly grows faster than '2 * n'
However:
Both '1000 * n' and '2 * n' are Θ(n)

What's the point of even having these complexity classes if I have functions with different growth rates in the same complexity class ?
If functions are in different complexity classes, we can tell which one grows faster without even knowing the constant.

e.g.
Any function in Θ(n^2) will grow faster than any function in Θ(n)
Even if we pick '0.000000000001 * n^2' and
'99999999999999999999 * n' the values from the first function will

eventually become larger than the values from the second function, and the difference between the two will increase

What does all of this have to do with logs ?

log_2(n), log_3(n), log_4(n), ... are all in the same complexity class Θ(log(n))
i.e log_a(n) is in Θ(log(n)) for all a > 1

Why are they all in the same complexity class ?
If we have f(n) = k * log_a(n) this is equivalent to:
f(n) = k * log(n)/log(a)
f(n) = (k/log(a)) * log(n)
(k/log(a) is just a constant so f(n) is Θ(log(n))

What is the difference in growth rates between logs with different bases ?

the larger the base, the slower it grows
e.g. log_8(n) grows slower than log_2(n)

Why ? Because the log is telling us what exponent on that base we would need to make that number.

A number like 8^n grows faster than 2^n.
So if we have 8^x = value = 2^y , we would expect that x would be smaller than y.
But all the log is doing is telling us what the exponent is for that value.
i.e. log_8(value)=x and log_2(value)=y
So we should expect that log_8(n) grow slower than log_2(n)

Here's a table which shows the difference in growth rates between log_8 and log_2

```
n=1             log_8(n)= 0  log_2(n)= 0
n=8             log_8(n)= 1  log_2(n)= 3
n=64            log_8(n)= 2  log_2(n)= 6
n=512           log_8(n)= 3  log_2(n)= 9
n=4096          log_8(n)= 4  log_2(n)= 12
n=32768         log_8(n)= 5  log_2(n)= 15
n=262144        log_8(n)= 6  log_2(n)= 18
n=2097152       log_8(n)= 7  log_2(n)= 21
n=16777216      log_8(n)= 8  log_2(n)= 24
n=134217728     log_8(n)= 9  log_2(n)= 27
```

Hope this makes sense

10 comments (81 votes) Flag more

See 5 more replies

Leon Gower 4 years ago

more

I'm following the content but i'm unsure as to why the content is heading down this path. What is the reasoning behind learning or even reviewing this? Shouldn't lessons about the various speeds of Algorithms come after we've learned the various algorithms? we were given a snippet of information regarding 2 simple algorithms and then sent into the woods with a hand full of bread crumbs... or did I miss something?

(15 votes) Flag more

Cameron 4 years ago

more

They use the asymptotic analysis when they discuss each algorithm, but if you wanted to you could:
-skip the asymptotic analysis section
-read the parts that discuss how each algorithm works (ignoring any asymptotic notation)
-skip the analysis of each algorithm
-after learning how each algorithm works, review the asymptotic notation section
-review the analysis section for each algorithm

The asymptotic notation is useful in telling the computer science part of the story. It tells you why one algorithm is better than another algorithm. It tells you why you would even bother learning merge sort and quick sort after learning about other sorting algorithms.

1 comment (11 votes) Flag more

Andrei Shindyapin 4 years ago

more

Θ(n!) would grow faster than Θ(2^n), correct?

(10 votes) Flag more

Cameron 4 years ago

more

To demonstrate that n! grows faster than 2^n, write them out like this:

```
n!    = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * ... * n
2^n   = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * ... * 2
```

Here's how we can show n! grows faster than 2^n:
If we have n! and 2^n written as above we can see that the ratio of the terms for n>2 is >1 and increasing with larger n.
That is:
(3 * 4 * 5 * ... * n)/(2 * 2 * 2 * ... * 2) >1 and increases with n.
Since this ratio is increasing, the ratio will become larger than the inverse ratio of the terms for n <=2.
That is:
(3 * 4 * 5 * ... * n)/(2 * 2 * 2 * ... * 2) will eventually become greater than (2 * 2)/(1 * 2)
Once that occurs, the ratio between all the terms in n! and 2^n will be >1
Not only will the ratio be >1, but it will also be increasing as n increases
This demonstrates that n! grows faster than 2^n
(Note that: The same logic can be used for n! and c^n where c is any constant)

Another approach is to use an approximation for n!.
Stirling's Approximation says:
n! ~= sqrt(2 * Pi * n) * (n/e)^n
(https://en.wikipedia.org/wiki/Stirling%27s_approximation)
From inspection we can see that even the:
(n/e)^n term grows faster than 2^n

4 comments (16 votes) Flag more

Hope this makes sense

See 1 more reply

nguyenhaitran97 4 years ago

more

what consider constant , linear ,polinomial, expotienal growth what is the rule for it?i havent learn log yet

(6 votes) Flag more

Yahaya Aluke 4 years ago

more

constant < logarithmic < linear < polynomial < exponential

1 comment (16 votes) Flag more

See 2 more replies

LuminatorBU 4 years ago

more

What is the difference between lg and log notation? Do they mean the same thing or are they different?

(4 votes) Flag more

Cameron 4 years ago

more

Here they are using:
- lg means log base 2
- log_a is log base a

The bad news:
- If you see log, or lg in other places it may mean something different.

e.g. log is often used to represent: log base 2, log base 10, log base e
e.g. lg is often used to represent: log base 2, log base 10, log base e
- Often, in computer science, log or lg will be assumed to be log base 2 without anyone mentioning it.

The good news:
- If you see Ln then you can be fairly sure it means the natural log
- If you see log_a or lg_a you can be fairly sure it means log base a
- In asymptotic analysis, the base of our logarithms usually don't matter, (because we can convert from one base to another by multiplying by a constant, and we can typically ignore these constants)

(6 votes) Flag more

See 2 more replies

justinj1776 4 years ago

more

Could you explain why a^n grows faster than b^n where a and b are constants.It might be possible for certain values of a and b but not for all.For example consider the case where:
1.a^n=100^n where a=100
Plot is here:https://www.wolframalpha.com/input/?i=100^n+from+n%3D0+to+100
2.n^b=n^100 where b=100
Plot is here:https://www.wolframalpha.com/input/?i=n^100+from+n%3D0+to+100

Looking at the plots I think step 2 increases faster than step 1 or am I wrong?

(2 votes) Flag more

Cameron 4 years ago

more

Perhaps taking the log of both equations will clarify things:
log (a^n) = n log a
log (n^b) = b log n
Now, as n increases, it should be clear that, the top equation is growing faster than the bottom one.

3 comments (10 votes) Flag more

See 3 more replies

William 4 years ago

more

The article mentions that "computer scientists like to think in powers of 2." I'm guessing this has something to do with the binary nature of computers, but is there a more significant meaning to this? As someone who is new to computer science, thinking in powers of 10 seems easier to me.

(2 votes) Flag more

Cameron 4 years ago

more

Modern computers work in binary, but there is a deeper meaning to it. Computer science is, to a large extent, built around the concept of Turing machines, which have a binary alphabet. (https://en.wikipedia.org/wiki/Turing_machine). The Church-Turing thesis (not proven) roughly says that if you can compute something, then it is computable on a Turing machine (see https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis), which is why it is used as the foundation of computer science.

Notably, when you see n used in functions for running times, the value of n represents the initial problem size as it would be represented on a Turing machine i.e. the size of the input data in binary. (This helps to make sure we are always comparing apples to apples)

Hope this makes sense

1 comment (6 votes) Flag more

Dídac Ruiç-Hernandez 3 years ago

more

Hello,
You explain quite well the case of a linear growth of an algorithm. It is actually quite intuitive. As well as it is the constant case. From here it is more or less straightforward to see how can an algorithm grow at a polynomial rate. However, I struggle to understand how can we see or conclude that the growth rate of an algorithm is logarithmic? Could you please give an example? Thanks

(2 votes) Flag more

Cameron 3 years ago

more

You get log_b(n) growth when you can reduce the problem size to 1/b of its previous size after each step.

Why ?
It will take log_b(n) steps to reduce the problem size to 1 (at which point it can be easily solved)
If each step requires only a constant amount of time, the running time will be O(log_b(n))

The classic example is binary search where you chop the #of possible locations in 1/2 after each guess, yielding a running time of O(log_2(n)).

Note: O(log_b(n)) is the same as O(log(n))

(4 votes) Flag more

See 2 more replies

Salem 2 years ago

more

Is there a negative correlation between an algorithm rate of growth and it's efficiency?

If my my expectation about the negative correlation is correct then the following statement are correct:
1- *Fast* rate of growth means *slow* algorithm. Therefore, *less* efficient algorithm.
2- *Slow* rate of growth means *fast* algorithm. Therefore, *more* efficient algorithm.

For example, in the linear search, the rate of growth is Θ(n), and the binary search, the rate of growth is Θ(lg n). In this example, the linear search has faster rate of growth than binary search which means that the linear search is less efficient than the binary search.

Is my expectation about the negative correlation correct?

(2 votes) Flag more

Cameron 2 years ago

more

Yes.
If you take less time to complete something, then you must have worked faster.
If you take more time to complete something, then you must have worked slower.

Just remember that asymptotic complexity only applies to large values of n. Simpler algorithms, which have worse asymptotic complexity often outperform complex algorithms with better asymptotic complexity when the values of n are small.
e.g.
Insertion sort is O(n^2), and merge sort is O(n log n).
For large values of n (> 100000) merge sort is faster than insertion sort.
For smaller values of n (<100) insertion sort is faster than merge sort.
This does not contradict anything that asymptotic complexity says, because asymptotic complexity only talks about what happens when we have large values of n.

(4 votes) Flag more

Show more...

‹ Big-θ (Big-Theta) notation

Comparing function growth ›