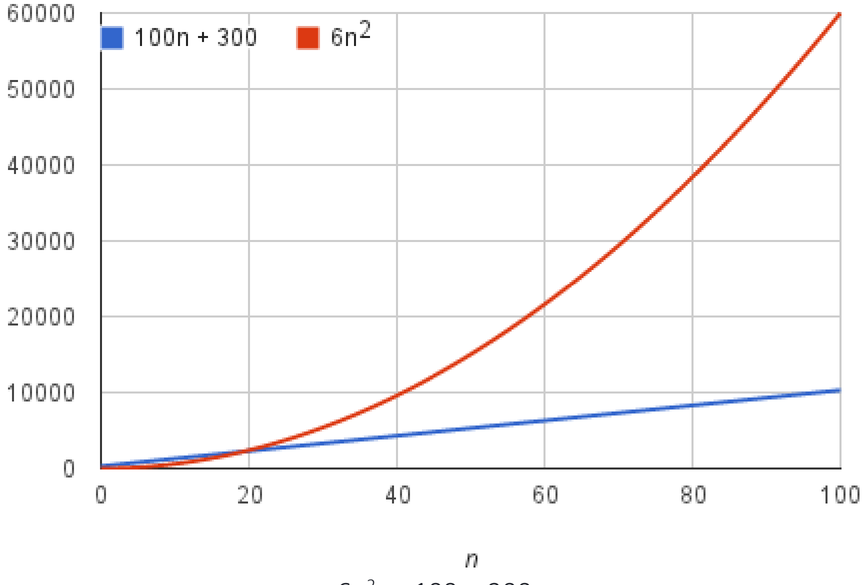
Asymptotic notation

The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm—and that depends on the speed of the computer, the programming language, and the compiler that translates the program from the programming language into code that runs directly on the computer, among other factors.

Let's think about the running time of an algorithm more carefully. We can use a combination of two ideas. First, we need to determine how long the algorithm takes, in terms of the size of its input. This idea makes intuitive sense, doesn't it? We know that the maximum number of guesses in linear search and binary search increases as the length of the array increases. Or think about a GPS. If it knew about only the interstate highway system, and not about every little road, it should be able to find routes more quickly, right? So we think about the running time of the algorithm as a *function of the size of its input*.

The second idea is that we must focus on how fast a function grows with the input size. We call this the **rate of growth** of the running time. To keep things manageable, we need to simplify the function to distill the most important part and cast aside the less important parts. For example, suppose that an algorithm, running on an input of size n, takes $6n^2 + 100n + 300$ machine instructions. The $6n^2$ term becomes larger than the remaining terms, $100n + 300$, once n becomes large enough, 20 in this case.

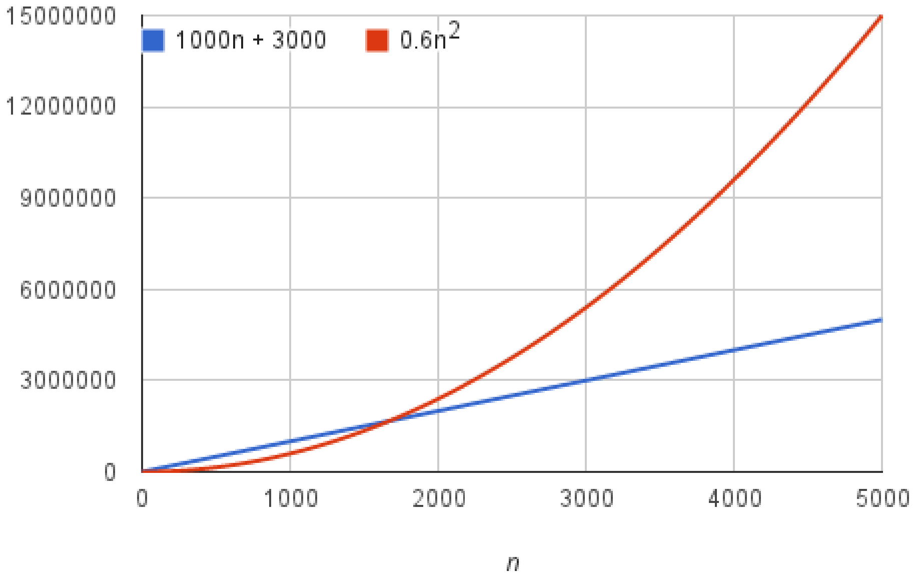
Here's a chart showing values of $6n^2$ and $100n + 300$ for values of n from 0 to 100:



$6n^2$ *vs* $100n+300$

We would say that the running time of this algorithm grows as n^2, dropping the coefficient 6 and the remaining terms $100n + 300$. It doesn't really matter what coefficients we use; as long as the running time is $an^2 + bn + c$, for some numbers $a > 0$, b, and c, there will always be a value of n for which an^2 is greater than bn + c, and this difference increases as n increases.

For example, here's a chart showing values of $0.6n^2$ and $1000n + 3000$ so that we've reduced the coefficient of n^2 by a factor of 10 and increased the other two constants by a factor of 10:



$6n^2$ *vs* $100n$+300

The value of n at which $0.6n^2$ becomes greater than $1000n + 3000$ has increased, but there will always be such a crossover point, no matter what the constants.

By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth—without getting mired in details that complicate our understanding. When we drop the constant coefficients and the less significant terms, we use **asymptotic notation**. We'll see three forms of it: **big-θ notation**, **big-O notation**, and **big-Ω notation**.

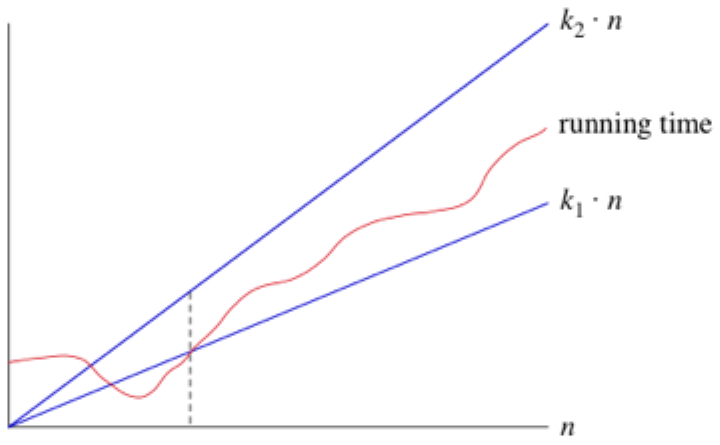
Big-θ (Big-Theta) notation

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

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

Here the size of the array is 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 length of the array (*n*);
compare array[guess] with targetValue; possibly return the value of guess; and 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 ***c1 · n***, where ***c1*** is the sum of the times for the computations in one loop iteration. Now, we cannot say here what the value of ***c1*** 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 ***c2***, which is also a constant. Therefore, the total time for linear search in the worst case is ***c1 · n + c2***.
As we've argued, the constant factor c1 and the low-order term c2 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 Θ(*n*). That's the Greek letter "theta," and we say "big-Theta of *n*" or just "Theta of *n*."

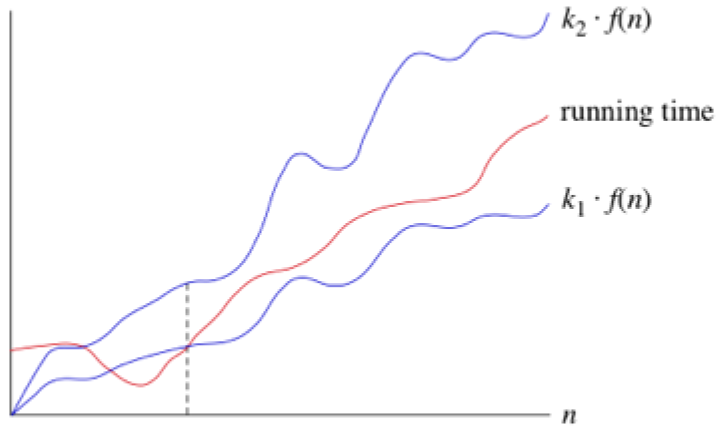
When we say that a particular running time is Θ(*n*), we're saying that once *n* gets large enough, the running time is at least ***k*1·*n*** and at most ***k*2·*n*** for some constants *k*1 and *k*2. Here's how to think of Θ(*n*):



$6n^2$ *vs* $100n+300$

For small values of n, we don't care how the running time compares with ***k*1·*n*** or ***k*2·*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 ·*n*** and ***k*2·*n***. As long as these constants *k*1 and *k*2 exist, we say that the running time is **Θ(*n*)**.

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



$6n^2$ *vs* $100n+300$

Once n gets large enough, the running time is between ***k*1·*f*(*n*)** and ***k*2·*f*(*n*)**.

In practice, we just drop constant factors and low-order terms. Another advantage of using big-Θ 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-Θ 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 **Θ(n^2)**.

When we use big-Θ 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.

Functions in asymptotic notation

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 $\Theta(lgn)$ time (that is, $\Theta(log_2 n)$ 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 *lgn+1*, generating and testing each guess takes constant time, and setting up and returning take constant time. As a matter of practice, we write that binary search takes $\Theta(lgn)$ time because computer scientists like to think in powers of 2 (and there are fewer characters to write than if we wrote $\Theta(log_2 n)$.)

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 $\Theta(n^{2.5})$.

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

Here's a list of functions in asymptotic notation that we often encounter when analyzing algorithms, listed from slowest to fastest growing. This list is not exhaustive; there are many algorithms whose running times do not appear here:

1. $\Theta(1)$
2. $\Theta(\lg n)$
3. $\Theta(n)$
4. $\Theta(n \lg n)$

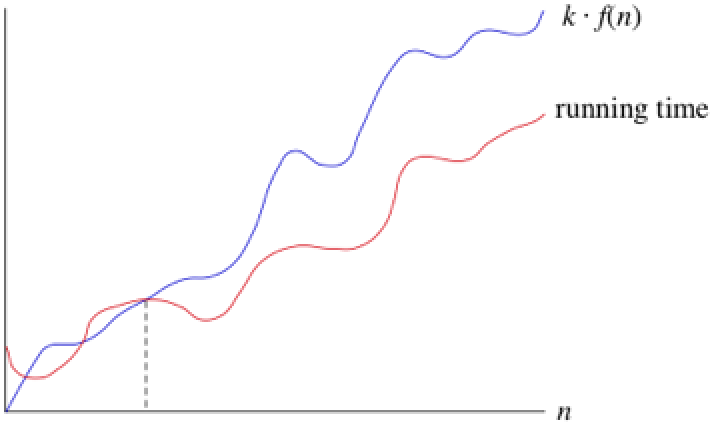
5. $\Theta(n^2)$
6. $\Theta(n^2\lg n)$
7. $\Theta(n^3)$
8. $\Theta(2^n)$

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

Big-O notation

We use big-Θ notation to asymptotically bound the growth of a running time to within constant factors above and below. Sometimes we want to bound from only above. For example, although the worst-case running time of binary search is Θ(lg*n*), it would be incorrect to say that binary search runs in Θ(lg*n*) time in *all* cases. What if we find the target value upon the first guess? Then it runs in Θ(1) time. The running time of binary search is never worse than Θ(lg*n*), but it's sometimes better. It would be convenient to have a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly." We use "big-O" notation for just such occasions.

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



$6n^2$ *vs* $100n$+300

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

Now we have a way to characterize the running time of binary search in all cases. We can say that the running time of binary search is always $O(\lg n)$. We can make a stronger statement about the worst-case running time: it's Θ(lg*n*). But for a blanket statement that covers all cases, the strongest statement we can make is that binary search runs in $O(\lg n)$ time.

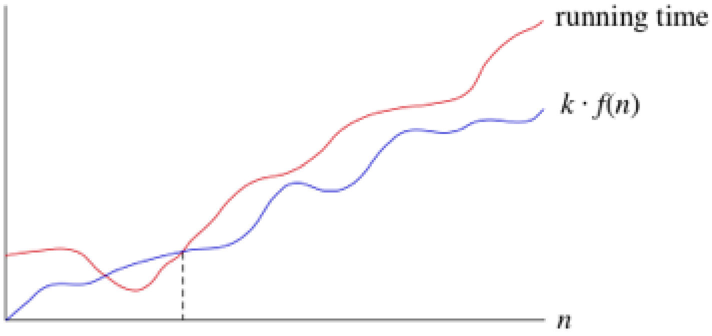
If you go back to the definition of big-Θ notation, you'll notice that it looks a lot like big-O notation, except that big-Θ notation bounds the running time from both above and below, rather than just from above. If we say that a running time is $\Theta(f(n))$ in a particular situation, then it's also $O(f(n))$. For example, we can say that because the worst-case running time of binary search is $\Theta(\lg n)$, it's also $O(\lg n)$. The converse is not necessarily true: as we've seen, we can say that binary search always runs in $O(\lg n)$ time but *not* that it always runs in $\Theta(\lg n)$ time.

Because big-O notation gives only an asymptotic upper bound, and not an asymptotically tight bound, we can make statements that at first glance seem incorrect, but are technically correct. For example, it is absolutely correct to say that binary search runs in $O(n)$. That's because the running time grows no faster than a constant times n. In fact, it grows slower. Think of it this way. Suppose you have 10 dollars in your pocket. You go up to your friend and say, "I have an amount of money in my pocket, and I guarantee that it's no more than one million dollars." Your statement is absolutely true, though not terribly precise. One million dollars is an upper bound on 10 dollars, just as $O(n)$ is an upper bound on the running time of binary search. Other, imprecise, upper bounds on binary search would be $O(n^2)$, $O(n^3)$, and $O(2^n)$. But none of $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$, and $\Theta(2^n)$ would be correct to describe the running time of binary search in any case.

Big-Ω (Big-Omega) notation

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 $\Omega(f(n))$:



$6n^2$ *vs* $100n+300$

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 $\Omega(\lg n)$. We can also make correct, but imprecise, statements using big-Ω notation. For example, just as 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," you can also say that the worst-case running time of binary search is Ω(1), because it takes at least constant time.

Some important points:

Why it is correct to say that the for the binary search the running time is Ω(lgn) and not Ω(1)?

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 Ω(g(n)) this means that f(n) grows asymptotically no slower than g(n)
if f(n) is Θ(g(n)) this means that f(n) grows asymptotically at the same rate as g(n)

As a result:
if f(n) is Θ(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 Ω(g(n)) and f(n) is O(g(n)).

Think of O as an upper bound, 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 neighbor.
We can immediately say that the height of our neighbor is upper bounded by 1 million kilometers.
We can also say that the height of our neighbor 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 neighbor at 5' 5", and an upper bound of 5' 10" we would have a much better idea of how tall our neighbor was.
If we had a lower bound of 5' 8" and an upper bound of 5' 8" we could then say that our neighbor is 5' 8".

So, for log(n) we could say:

log(n) is O(n^2) since log(n) grows asymptotically no faster than n^2
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 Ω(1) since log(n) grows asymptotically no slower than 1
log(n) is Ω(log(log(n))) since log(n) grows asymptotically no slower than log(log(n))
log(n) is Ω(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 Ω(log(n)) we can say that log(n) is Θ(log(n)).

Does Θ(g(n)) only exist if O(g(n)) and Ω(g(n)) have the same growth rate?

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 <= 150n^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 >= 0.1log(n)
Then I can say f(n) is Ω(log(n))

At this point, my analysis isn't good enough for me to make a statement about Θ(g(n))
I would only be able to tell you that f(n) is Θ(g(n)) if my analysis showed that f(n) is O(g(n)) and f(n) is Ω(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 <= 100$n^{1.5}$ + 1000 i.e. O($n^{1.5}$)
- f(n) is >= 10$n^{1.5}$ + 2 i.e. Ω($n^{1.5}$).

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

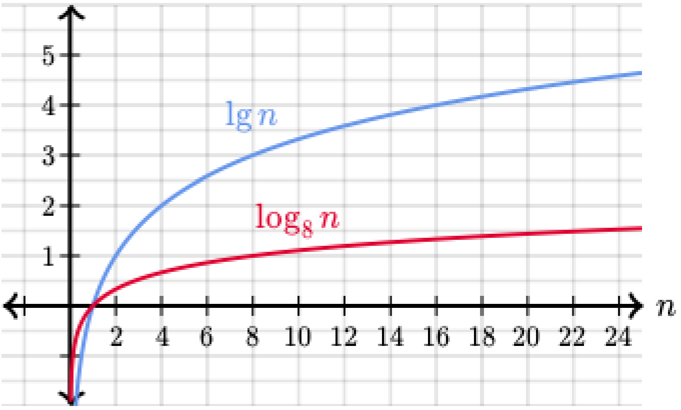
However, if I wasn't able to improve my analysis , and I still only knew that f(n) is O(n^2) and f(n) is Ω(log (n)), I wouldn't be able to give you a statement about Θ(g(n)).
However, it is trivially true that f(n) is O(f(n)) and that f(n) is Ω(f(n)) and thus f(n) is Θ(f(n)) i.e. we may not be clever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

Some practice questions:

For the functions, lg*n* and $\log_8 n$, what is the asymptotic relationship between these functions?

To answer this, we need to think about the function, how it grows, and what functions bind its growth.

Both log$_2$*n*, and log$_8$*n* are functions with logarithmic growth, with their base as the only difference. Here's a graph of the two functions:



If we were to say that lg*n* is *O*(log$_8$*n*), then we would be saying that lg*n* has an asymptotic upper bound of *O*(log$_8$*n*), i.e. that for large enough *n*, the running time is at most *k*·log$_8$*n*, for some constant *k*.

Can we come up with a *k* to multiply log$_8$*n* by so that it always bounds lg*n*?

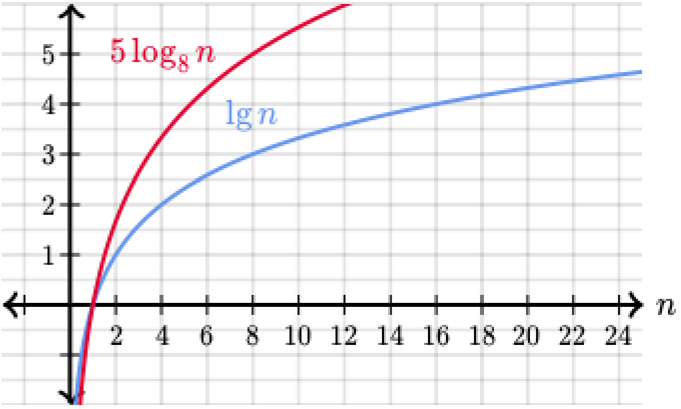
Yes, we can. It helps to remember this property of logarithms:

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

That property means that log$_8$*n* can be rewritten in terms of lg*n*:

$$\log_8 n = \frac{\lg n}{\lg 8}$$

We can now see that log$_8$*n* is just lg*n* times a constant. When we find two functions differ by a constant multiplier, then we know we can always find a *k* to serve as the upper bound.
For example, here's a graph where *k*=5:



Generally, we can effectively ignore constants in asymptotic notation, and treat the functions as equivalent.

All of that means that yes, indeed, lg$n$ is $O(\log_8 n)$.

If we were to say that lg$n$ is $\Omega(\log_8 n)$, then we would be saying that lg$n$ has an asymptotic lower bound of $\Omega(\log_8 n)$ - i.e. that for large enough n, the running time is at least $k \cdot \log_8 n$, for some constant k.

Can we come up with a k for to multiply $\log_8 n$ by to make that true?

As we went through above, we were able to rewrite $\log_8 n$ as lg$n$ multiplied by a constant:

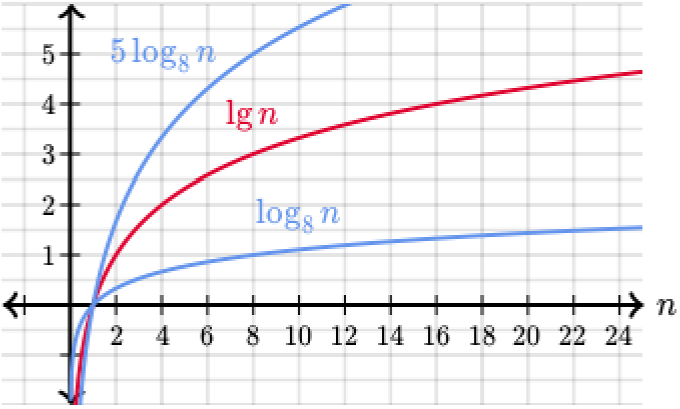
$$\log_8 n = \frac{1}{\lg 8} \lg n$$

Because that constant is a fraction, it already satisfies the k criteria, and we could see from the original graph how lg$n$ was always at least that value, for n > 1.

That means that yes, lg$n$ is $\Omega(\log_8 n)$.

If we were to say that lg$n$ is $\Theta(\log_8 n)$, then we would be saying that lg$n$ is "tightly bound" by $\Theta(\log_8 n)$ - i.e. that for large enough n, the running time is at least $k1 \cdot \log_8 n$ and at most $k2 \cdot \log_8 n$ for some constant $k1$ and $k2$. In fact, we already found a $k1$ and a $k2$ that satisfies those constraints, when determining the O and Ω notation.

Here's a graph where $k1$=1 and $k2$=5:



Since we established lg$n$ is $O(\log_8 n)$ and $\Omega(\log_8 n)$, then we can conclude that it is also $\Theta(\log_8 n)$.