L2- Analysis Framework + Notations

Textbook Sections 2.1-2.2

Outline

In Section 2.1, we talk about run time and algorithm speed

In Section 2.2, we introduce three notations: O (“big oh”), Ω (“big omega”), and Θ (“big theta”).

Efficiencies from Section 1.2

We already mentioned in Section 1.2 that there are two kinds of efficiency: **time efficiency and space efficiency**.

Efficiencies from Section 1.2

We already mentioned in Section 1.2 that there are two kinds of efficiency: **time efficiency and space efficiency.** Time efficiency, also called time complexity, indicates how fast an algorithm in question runs. Space efficiency, also called space complexity, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

Measuring an Input's Size

Almost all algorithms run longer on larger inputs. It takes longer to sort larger arrays, multiply larger matrices, and so on. In most cases, selecting such a parameter is quite straightforward. For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists.

This section in the textbook can be confusing so let's give some examples to clarify:

Measuring an Input’s Size

If the input is the type that changes size, let say a todo list, then we can let n be the amount of things in the todo list.

If the input is the type that doesn’t change size, let say a phone number, there is only one type of phone number which is constant, everybody has 10 numbers as their phone number within the country. then we can say that the input is of constant size no matter what.

Units for Measuring Run Time

One possible approach is to count the number of times each of the algorithm's operations is executed.

A better approach is to identify the most important operation of the algorithm, called the **basic operation**, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

Summary

- Both time and space efficiencies are measured as functions of the algorithm’s input size.
- Time efficiency is measured by counting the number of times the algorithm’s basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
- The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.
- The framework’s primary interest lies in the order of growth of the algorithm’s running time (extra memory units consumed) as its input size goes to infinity.

Let’s play a game

Let's play a little game to give you an idea of how different algorithms for the same problem can have wildly different efficiencies. I am going to randomly select an integer from 1 to 15. One student will guess numbers until he/she finds my number. (I will tell you each time if your guess was too high or too low)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Let’s write some code to show Binary search

Python code

Binary search vs Divide and Conquer

We will get to divide and conquer in a later chapter, but I will add some discussion on this..

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

Who can help write some pseudo code for binary search?

Volunteer

Here is a simplified version (B)

1. Let min = 1 and max = n
2. Guess the average of max and min rounded down so that it is an integer.
3. If you guessed the number, stop. You found it!
4. If the guess was too low, set min to be one larger than the guess.
5. If the guess was too high, set max to be one smaller than the guess.
6. Go back to step two

Here is a more correct version (A)

1. Let min = 0 and max = n-1.
1. If max < min, then stop: target is not present in array. Return -1.
2. Compute guess as the average of max and min, rounded down (so that it is an integer).
3. If array[guess] equals target, then stop. You found it! Return guess.
4. If the guess was too low, that is, array[guess] < target, then set min = guess + 1.
5. Otherwise, the guess was too high. Set max = guess - 1.
6. Go back to step 2.

From step 2: “if max < min, then stop” ??

This step will check for if the number is in the array. In our guessing game this wasn’t necessary, but in a “real” algorithm, a check step is absolutely necessary

The target number isn't in the array if there are no possible guesses left.

Let’s use an array of prime numbers as an example:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97];
```

Run time for binary search

We know that binary search runs faster than linear. Reminder of why?

Are you seeing a pattern in the number of guesses needed to guess an integer from an array of length n?

Run time for binary search

Every time we double the size of the array, we need at most one more guess.

Suppose we need at most m guesses for an array of length n. Then, for an array of length 2n the first guess cuts the reasonable portion of the array down to size n and at most m guesses finish up, giving us a total of at most m+1 guesses.

What equation can be used to represent this?

Run time for binary search

Fortunately, there's a mathematical function that means the same thing as the number of times we repeatedly halve, starting at n until we get the value 1: the base-2 logarithm of n. That's most often written as $\log_2 n$

What if n isn't a power of 2?

Run time for binary search

Fortunately, there's a mathematical function that means the same thing as the number of times we repeatedly halve, starting at n until we get the value 1: the base-2 logarithm of n. That's most often written as $\log_2 n$

What if n isn't a power of 2? In that case, we can look at the closest lower power of 2. For an array whose length is 1000, the closest lower power of 2 is 512, which equals 2^9. We can thus estimate that $\log_2 1000$ is a number greater than 9 and less than 10, or use a calculator to see that it’s about 9.97. Adding one to that yields about 10.97. In the case of a decimal number, we round down to find the actual number of guesses. Therefore, for a 1000-element array, binary search would require at most 10 guesses.

Orders of Growth

So far, we analyzed linear search and binary search by counting the maximum number of guesses we need to make. But what we really want to know is how long these algorithms take. We're interested in time, not just guesses.

We think about the running time of the algorithm as a function of the *size of its input.*

Orders of Growth

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've already seen 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?

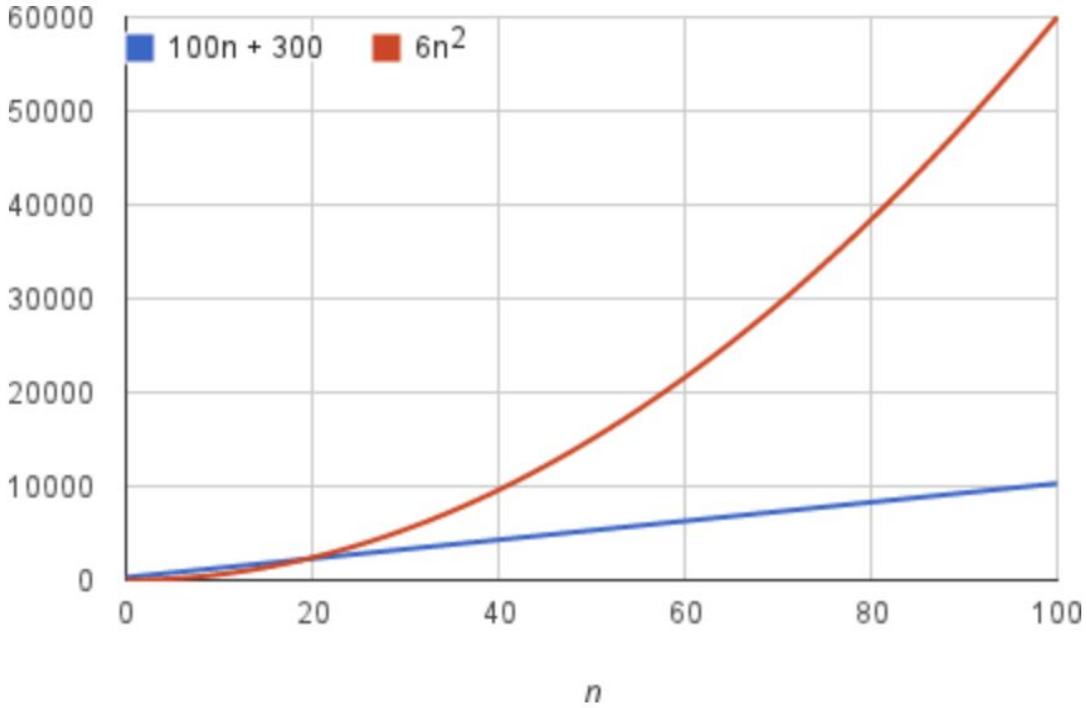
Orders of Growth

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

Orders of Growth

Here is a chart showing values of $6n^2$ and $100n + 300$ of n from 0 to 100



Asymptotic Notation

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:

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

Big-θ (Big-Theta) notation

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

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

Big-θ (Big-Theta) notation

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

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

1. Compare guess with array.length
2. Compare array[guess] with targetValue
3. Possibly return the value of guess
4. Increment guess

Big-θ (Big-Theta) notation

Each of these little computations takes a constant amount of time each time it executes. If the for-loop iterates n times, then the time for all n iterations is $c_1 * n$ where c_1 is the sum of the times for the computations in one loop iteration.

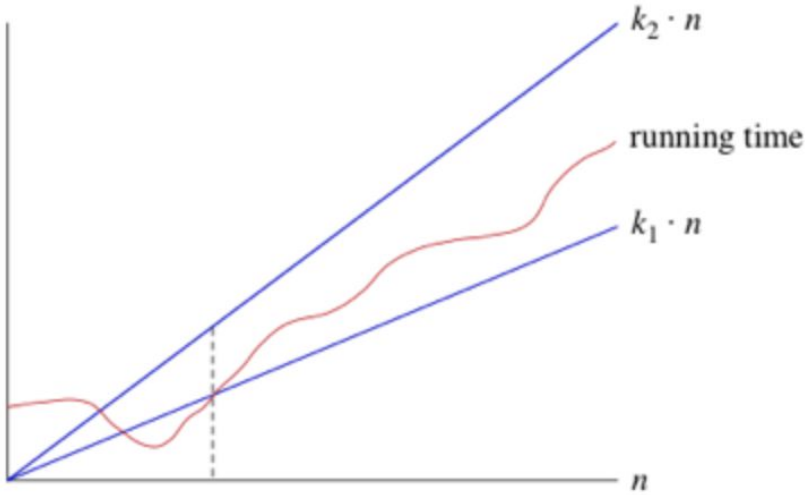
Big-θ (Big-Theta) notation

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

As we've argued, the constant factor c1 and the low-order term c_2 don't tell us about the rate of growth of the running time. What's significant is that the worst-case running time of linear search grows like the array size n. The notation we use for this running time is Θ(n)

Big-θ (Big-Theta) notation

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)



Big-θ (Big-Theta) notation

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 $\Theta(n^2)$

Big-θ (Big-Theta) notation

What do I mean by “tight-bound”

Hint: think of the definition of “asymptotic”

Big-θ (Big-Theta) notation

What do I mean by “tight-bound”

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

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

Summary

Θ(f(n)): The set of functions that grows no faster and no slower than f(n)

- asymptotic tight-bound on growth rate

Asymptotic Notation

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 $\Theta(n^0)$
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 $\Theta(1)$.

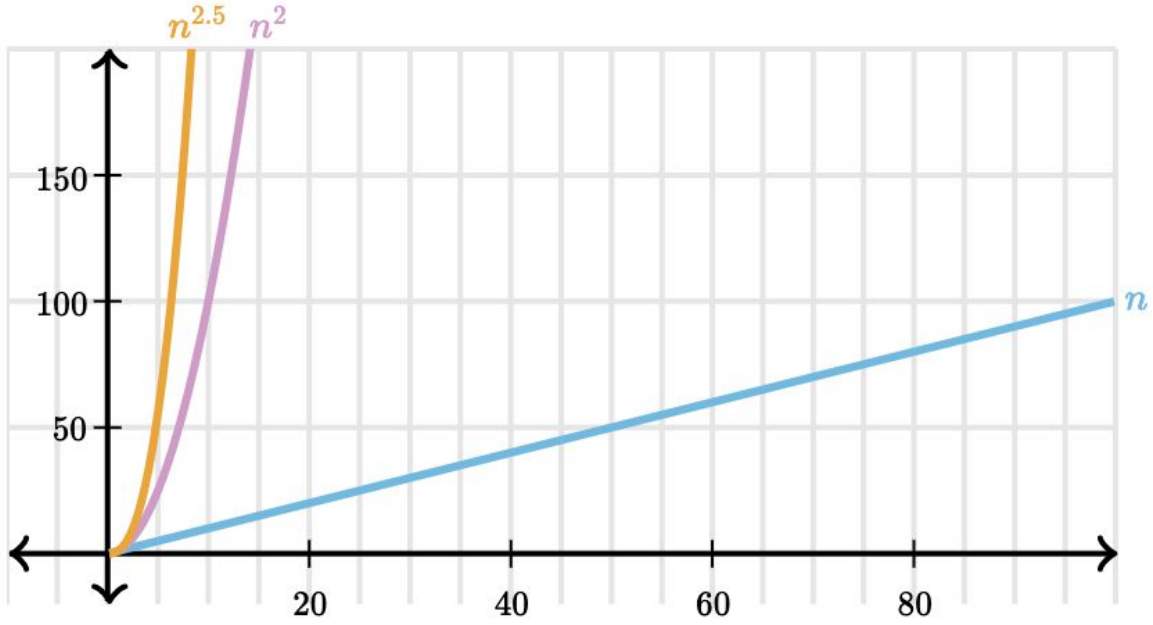
Asymptotic Notation

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 $\Theta(1)$

Let’s look at some graphs

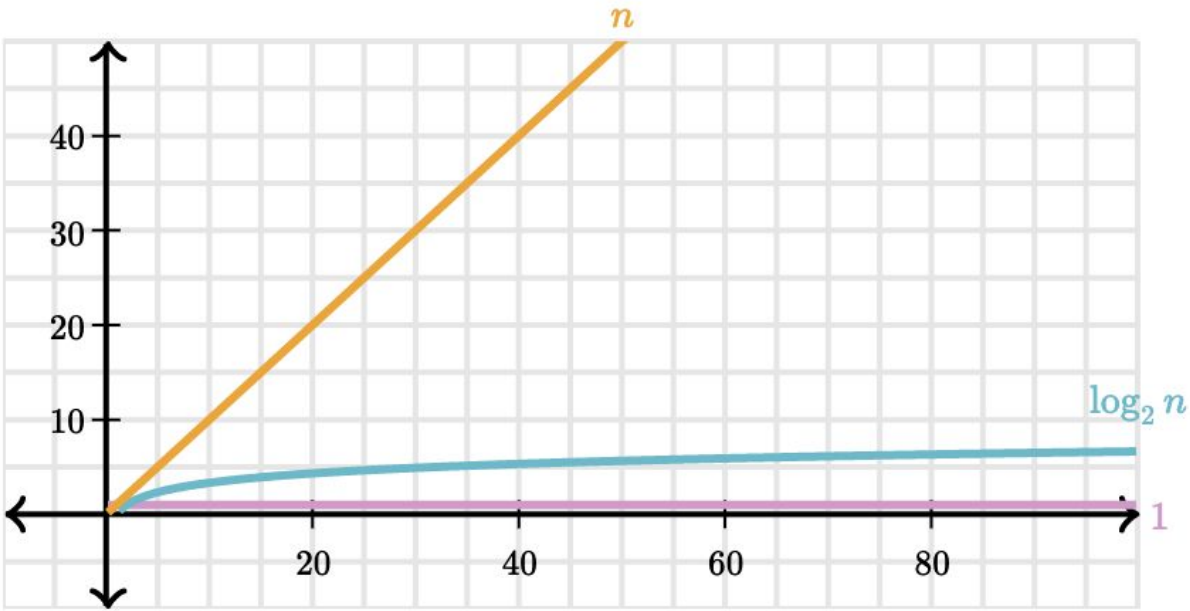
Asymptotic Notation

The following graph compares the growth of n (blue), n^2 (pink), and $n^{2.5}$ (orange)



Asymptotic Notation

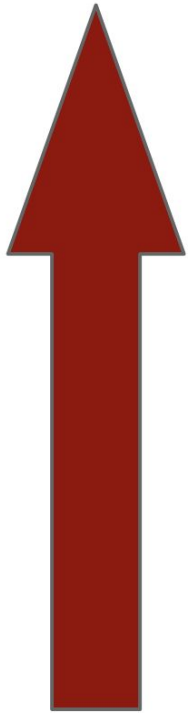
The following graph compares the growth of 1 (purple), n (orange), and $\log_2 n$ (blue)



Asymptotic Notation

Here is a growth rate ranking of typical functions:

$$f(n) = n^n$$
$$f(n) = 2^n$$
$$f(n) = n^3$$
$$f(n) = n^2$$
$$f(n) = n \log n$$
$$f(n) = n$$
$$f(n) = \sqrt{n}$$
$$f(n) = \log n$$
$$f(n) = 1$$



grow fast

grow slowly

Practice: Comparing Function Growth

Rank these functions according to their growth, from slowest growing to fastest growing.

$8n^2$

$n \log_6 n$

64

$\log_2 n$

$n \log_2 n$

$\log_8 n$

$6n^3$

$4n$

8^{2n}

Practice: Comparing Function Growth

The answer is:

64

$\log_8 n$

$\log_2 n$

$4n$

$n \log_6 n$

$n \log_2 n$

$8n^2$

$6n^3$

8^{2n}

Practice: Comparing Function Growth

Rank these functions according to their growth,
from slowest growing (at the top)
to fastest growing (at the bottom).

1

n^3

n^2

$(3/2)^n$

n

2^n

Practice: Comparing Function Growth

The answer is:

1

n

n^2

n^3

$(3/2)^n$

2^n

Explanation

1. Constant functions
    - 1
2. Linear functions
    - n
3. Polynomial functions
    - n^2, n^3
4. Exponential functions
    - 2^n, $(3/2)^n$

Practice: Comparing Function Growth

What kind of growth characterizes each function?

	Constant	Linear	Polynomial	Exponential
$2n^3$	○	○	○	○
1000	○	○	○	○
$(3/2)^n$	○	○	○	○
2^n	○	○	○	○
$3n$	○	○	○	○
1	○	○	○	○
$(3/2)n$	○	○	○	○
$3n^2$	○	○	○	○

Practice: Comparing Function Growth

The answer is:

	Constant	Linear	Polynomial	Exponential
$2n^3$	○	○	●	○
1000	●	○	○	○
$(3/2)^n$	○	○	○	●
2^n	○	○	○	●
$3n$	○	●	○	○
1	●	○	○	○
$(3/2)n$	○	●	○	○
$3n^2$	○	○	●	○

Practice: Comparing Function Growth

Match each function with an equivalent function, in terms of their Θ. Only match a function if f(n)=Θ(g(n))

.

$f(n)$	$g(n)$
$n + 30$	n^4
$n^2 + 2n - 10$	$3n - 1$
$n^3 * 3n$	$\log_2 2x$
$\log_2 x$	$n^2 + 3n$

Practice: Comparing Function Growth

The answer is:

$f(n)$	$g(n)$
$n + 30$	$3n - 1$
$n^2 + 2n - 10$	$n^2 + 3n$
$n^3 * 3n$	n^4
$\log_2 x$	$\log_2 2x$

Explanation

$f(n)=\Theta(g(n))$ if there exists constants c1, c2 >0, such that for all sufficiently large n, we have $c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)$

To figure out which of the expressions have that property, **we can start by dropping the constants and lower order terms**. We can drop the constants because we know that we can just pick any constant to multiply by to satisfy the above constraint anyway. We can drop lower order terms because for sufficiently large n, we know that lower order terms never contribute as much to the rate of growth.

Explanation

Here are the f(x) expressions with their constants dropped and lower order terms removed:

$f(x)$	Simplified
$n^2 + 2n - 10$	n^2
$n^3 * 3n$	n^4
$n + 30$	n
$\log_2 x$	$\log_2 x$

Explanation

Here are the g(x) expressions with their constants dropped and lower order terms removed:

$g(x)$	Simplified
$n^2 + 3n$	n^2
n^4	n^4
$3n - 1$	n
$\log_2 2x$	$\log_2 x$

Explanation

Now that we've simplified the expressions, it should be easy to match them:

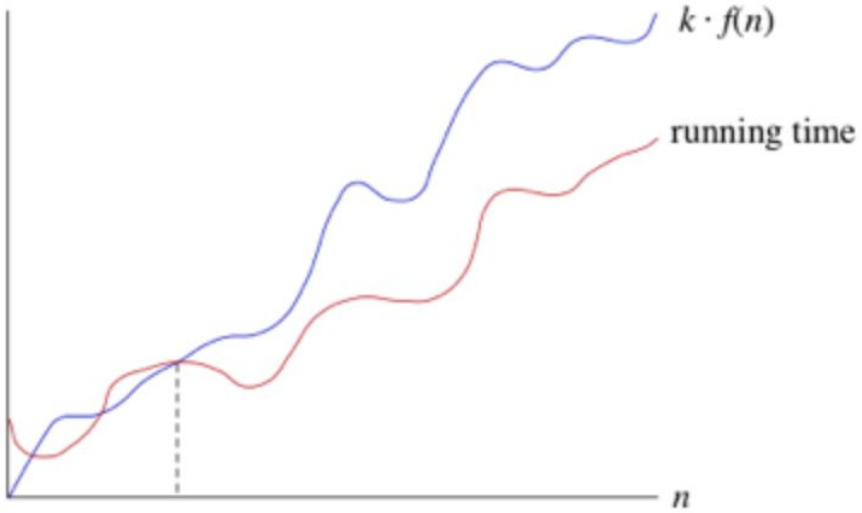
$f(x)$	Simplified	$g(x)$	Simplified
$n^2 + 2n - 10$	n^2	$n^2 + 3n$	n^2
$n^3 * 3n$	n^4	n^4	n^4
$n + 30$	n	$3n - 1$	n
$\log_2 x$	$\log_2 x$	$\log_2 2x$	$\log_2 x$

Big-O (Big-Oh) 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.

Big-O (Big-Oh) notation

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



Big-O (Big-Oh) notation

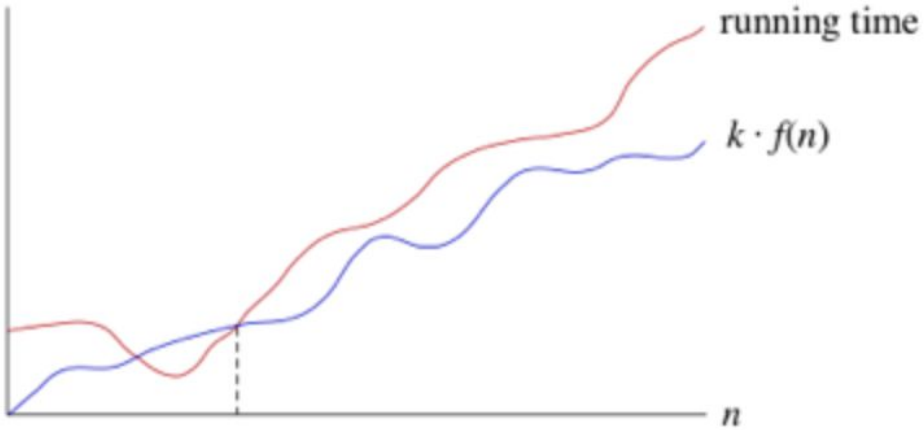
Because big-O notation gives only an asymptotic upper bound, and not an asymptotically tight bound, we can make statements that at first glance seem incorrect, but are technically correct. For example, it is absolutely correct to say that binary search runs in O(n) time. That's because the running time grows no faster than a constant times n. In fact, it grows slower.

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."

Big-Ω (Big-Omega) notation

If a running time is Ω(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 Ω(f(n))



Big-Ω (Big-Omega) notation

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 Ω(1), because we know that it takes at least constant time.

Practice: Asymptotic Notation

For the functions 8^n and 4^n
what is the asymptotic relationship
between these functions?

Choose all answers that apply:

- (A) 8^n is $O(4^n)$
- (B) 8^n is $\Omega(4^n)$
- (C) 8^n is $\Theta(4^n)$

Practice: Asymptotic Notation

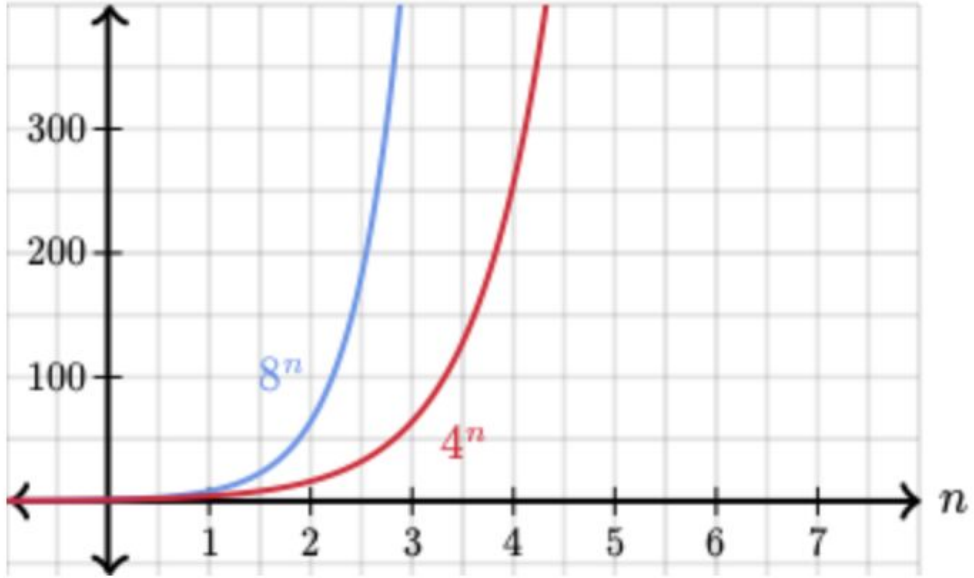
The answer is:

Choose all answers that apply:

- (A) 8^n is $O(4^n)$
- (●) 8^n is $\Omega(4^n)$
- (C) 8^n is $\Theta(4^n)$

Practice: Asymptotic Notation

Let’s look at a graph for help



In Summary

If you have a function f(N):

Big-O tells you which functions grow at a rate >= than f(N), for large N

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

Big-Omega tells you which functions grow at a rate <= than f(N), for large N

(Note: >= , "the same", and <= are not really accurate here, but the concepts we use in asymptotic notation are similar):

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