Announcements

January 28, 2015

- Lab 2 due today…
- Lab 3 out today!
  - Due Wednesday (2/4/15)
- SA-6 will likely go out tomorrow (possibly Friday)
  - Due Monday (2/2/15)
- X-hour tomorrow

Review of Analysis of Algorithms

Where we've been…

- Covered Linked Lists
  - Double Linked Lists w/ Sentinel
  - Singly Linked Lists
  - Growing Arrays (ArrayLists)
- Discussed various operations (add, remove, search)
  - Generally easier with DLLs w/ Sentinel…
- Discussed variations on SLL implementations (Lab 3!)

Where we are going…

- "Quick" (one lecture) overview / review of the basic ways to characterize algorithmic complexity
- Chapter 4 of our textbook covers some great details
- CS 31 and CS 231 go even deeper!
- Today
  - Orders of growth
  - Big O, Θ, and Ω
  - Working w/ asymptotic notation

Why we are going here…

- The book addresses the idea of “experimental studies”
- Short comings of this kind of experimental analysis:
  - variance in hardware can impact observations…
  - can only conduct experiments on a limited set of test inputs…
  - an algorithm must be fully implemented in order to execute and study it…
- **We want:**
  - an approach for evaluating *relative* efficiency independent of HW/SW.
  - to consider all possible inputs
  - to study an algorithm at a high-level (w/out having to implement)

Orders of Growth

Order of Growth

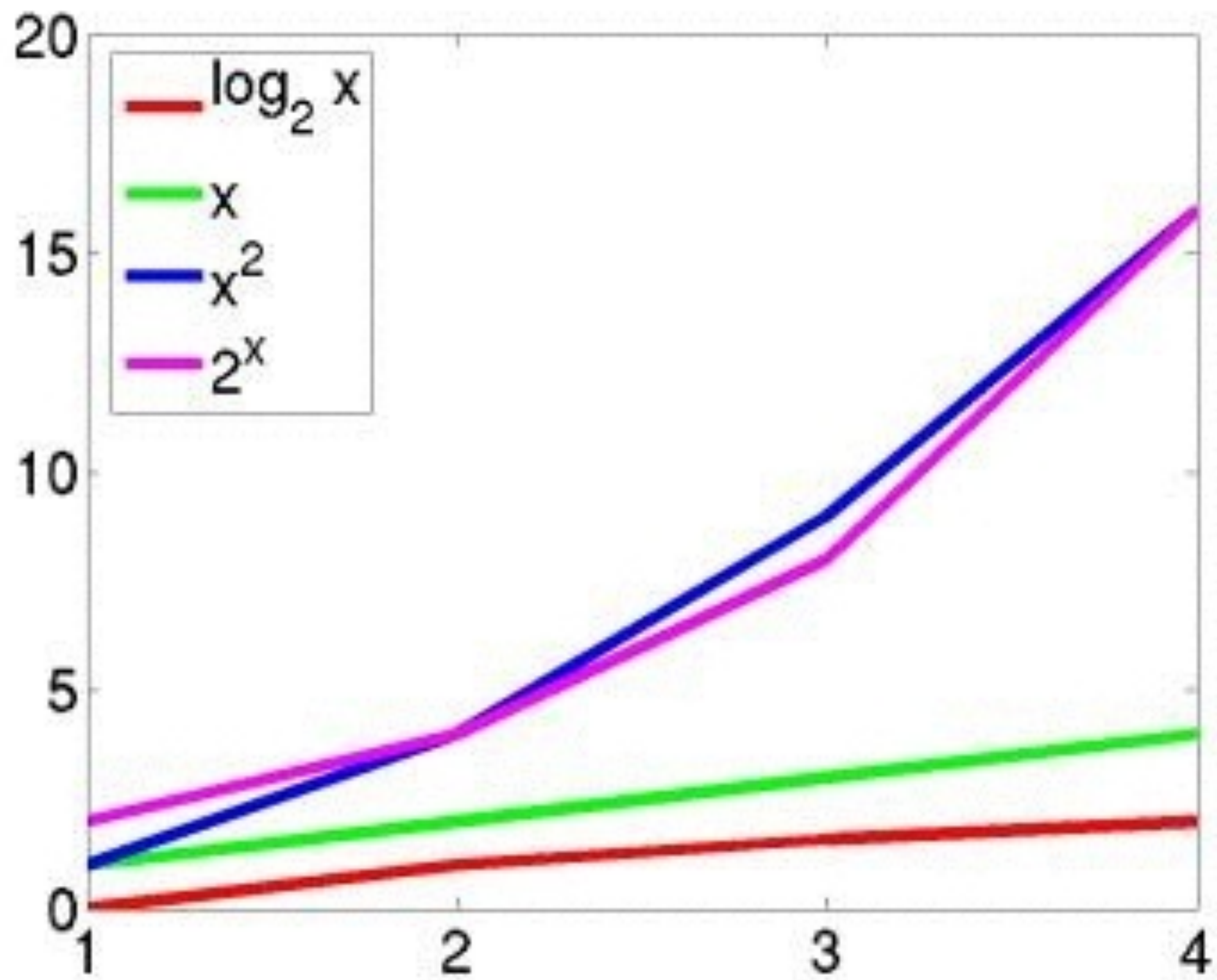
- Some (hopefully) familiar functions:

$$f(x) = x$$
$$f(x) = x^2$$
$$f(x) = log(x) \text{ and } f(x) = ln(x)$$
$$f(x) = 2^x$$

- The book addresses more functions than the ones we will look at in class and covers this in greater detail; after class you should have the necessary tools/understanding to go to the text and navigate those examples.

Order of Growth

(1)

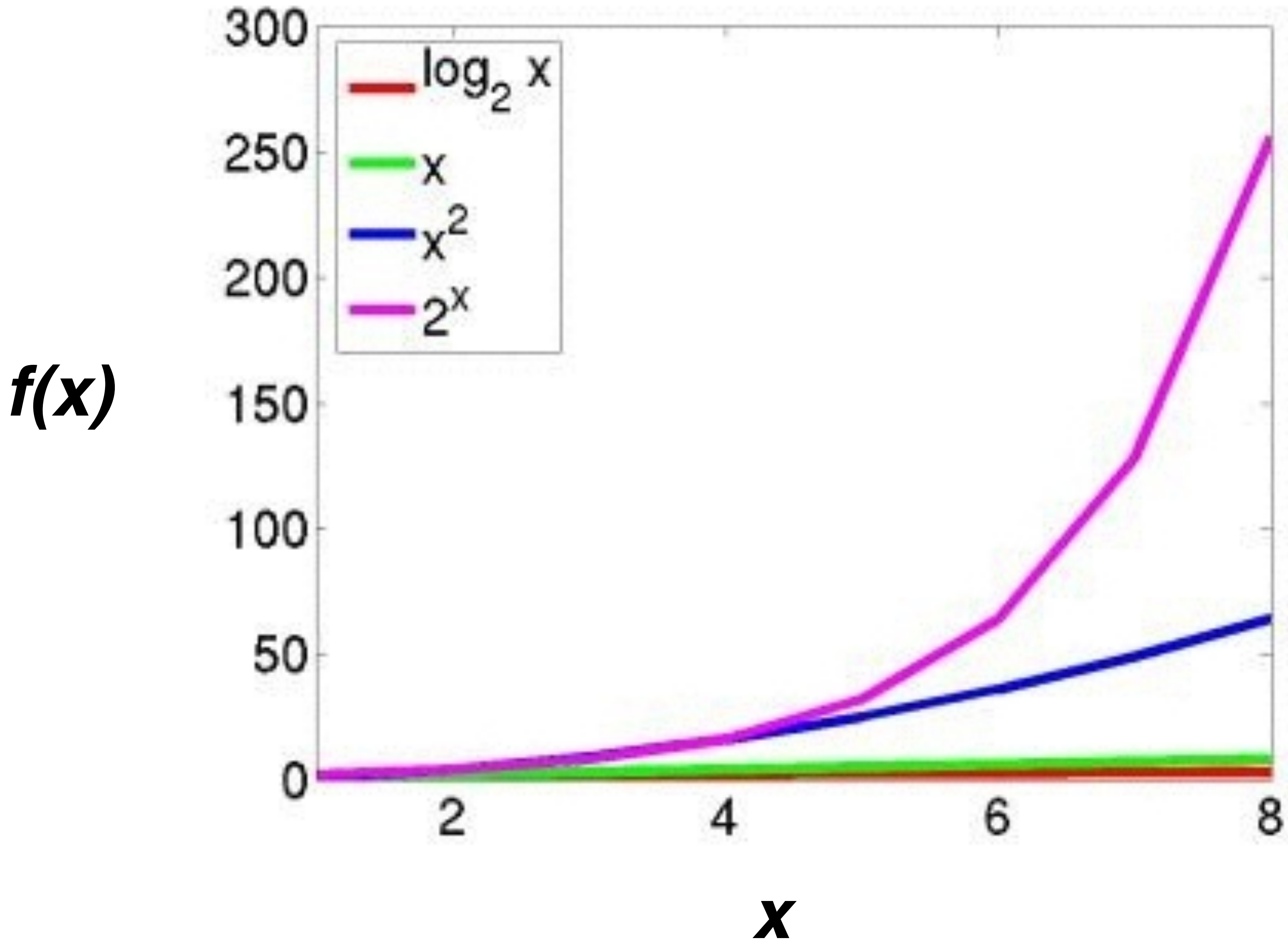


f(x)

x

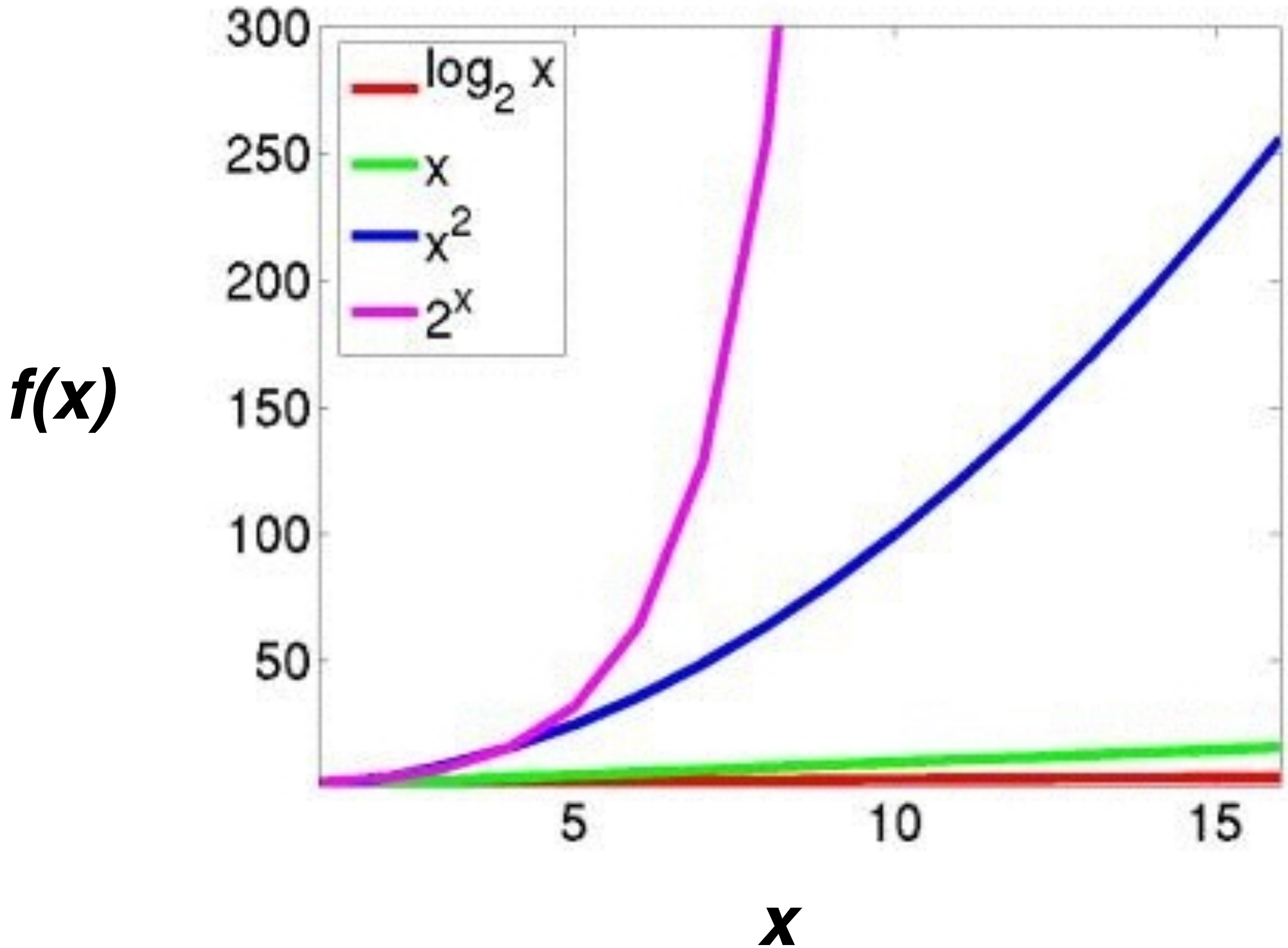
Order of Growth

(2)



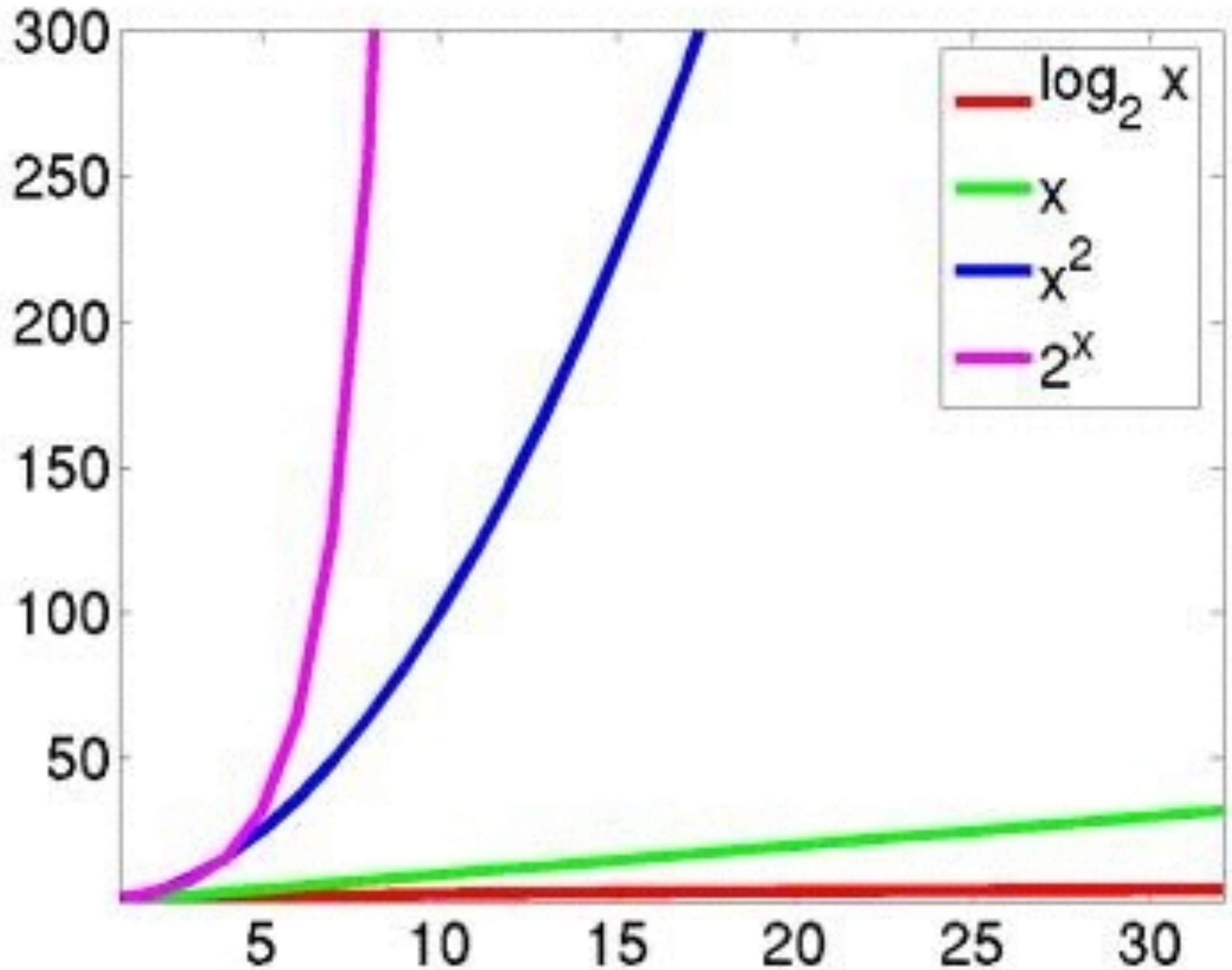
Order of Growth

(3)



Order of Growth

(4)

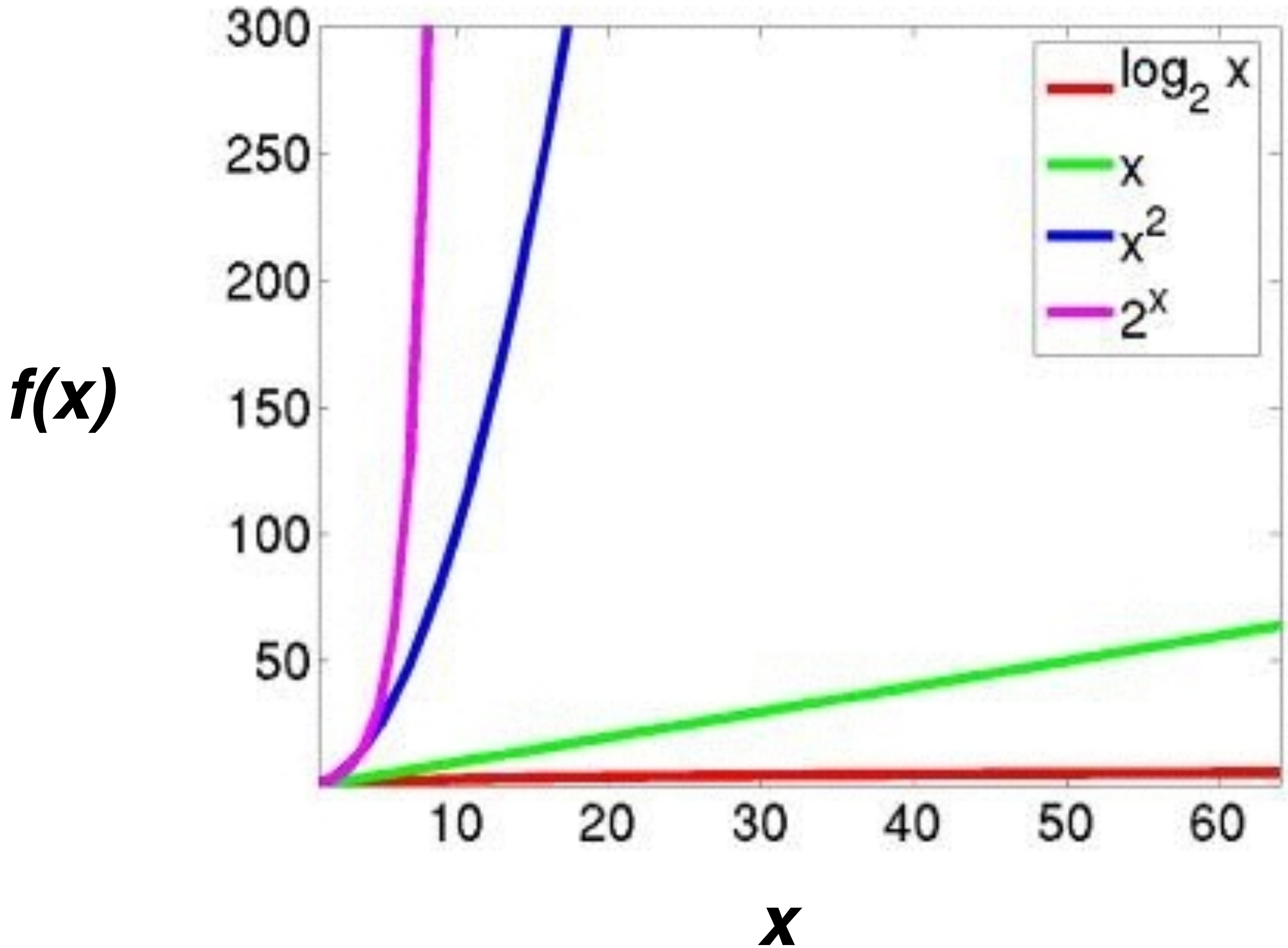


f(x)

x

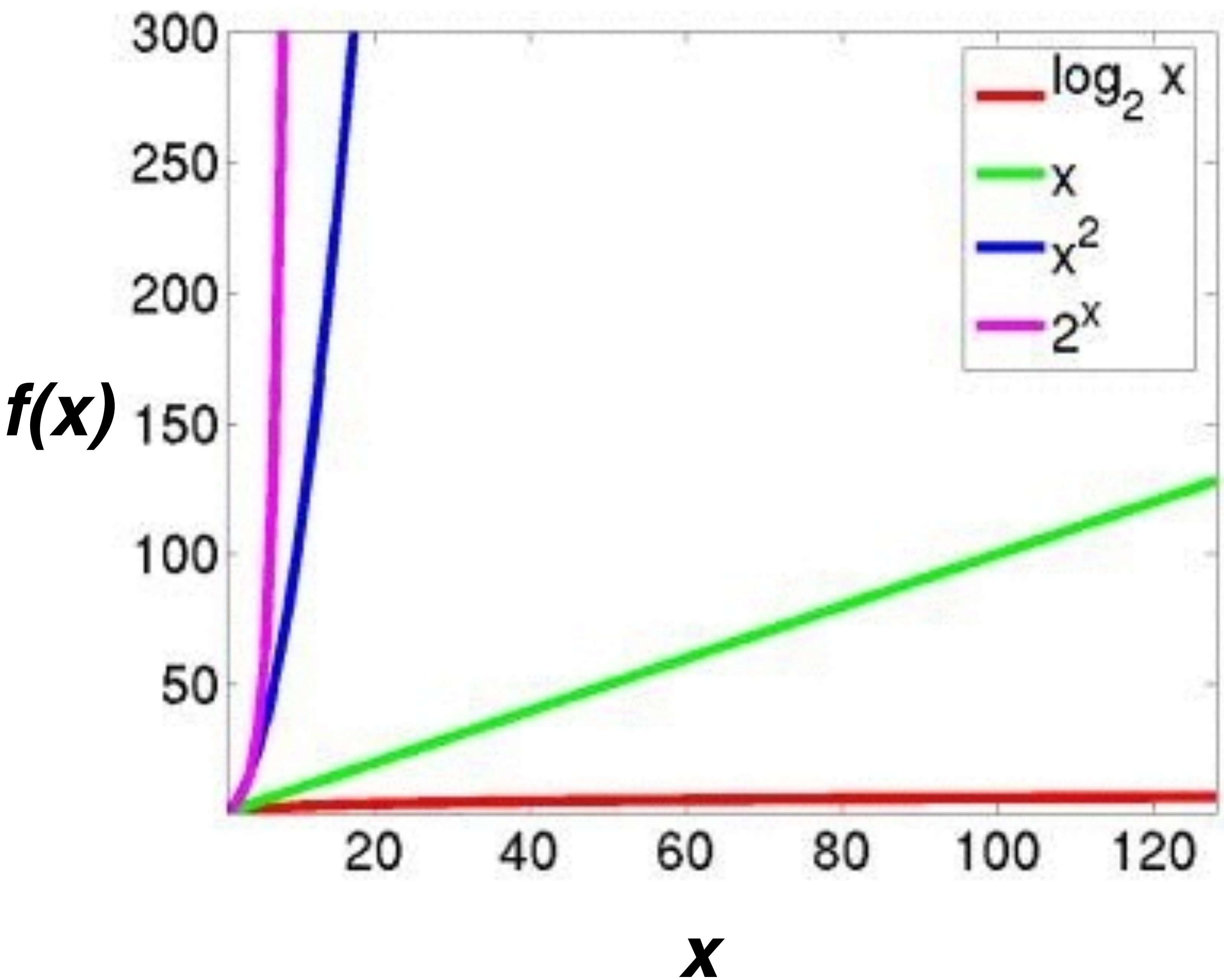
Order of Growth

(5)



Order of Growth

(6)



NOTE: That pretty purple line is why you can use the web safely :)

When you visit sites that have an **https://** (rather than http://) a special security algorithm (RSA) is working "under the hood" to establish secure/trusted connections between your computer and some server.

The RSA algorithm uses a "really hard" math problem (prime number factoring of BIG numbers) to protect things like passwords, etc.

Cracking passwords involves solving the prime factorization problem which runs proportional to exponential time.

Order of Growth

Computers are always getting faster because hardware is always changing…

"Orders of growth" help us see at a glance the inherent differences in run-time for different algorithms.

Supposing a computer could do a single operation in **0.0001** seconds, we'd have the following total amounts of time, for various problem sizes and various orders of growth.

order	**10**	**50**	**100**	**1000**
$log(n)$	0.0003 s	0.0006 s	0.0007 s	0.001 s
n	0.001 s	0.005 s	0.01 s	0.1 s
n^2	0.01 s	0.25 s	1 s	1.67 min
2^n	0.1024 s	3570 yrs	4×10^{18} yrs	forget about it

Big O, Θ, and Ω

Big O, Θ, and Ω

- Motivations
  - The notion of "grows like" is the essence of the running time
  - In the general case, we have some input and some mathematical function. In terms of running time, we consider the *size* of the input and the time it takes for some function (algorithm) to compute over the input.
- Assumptions
  - can drop coefficients and low-order terms

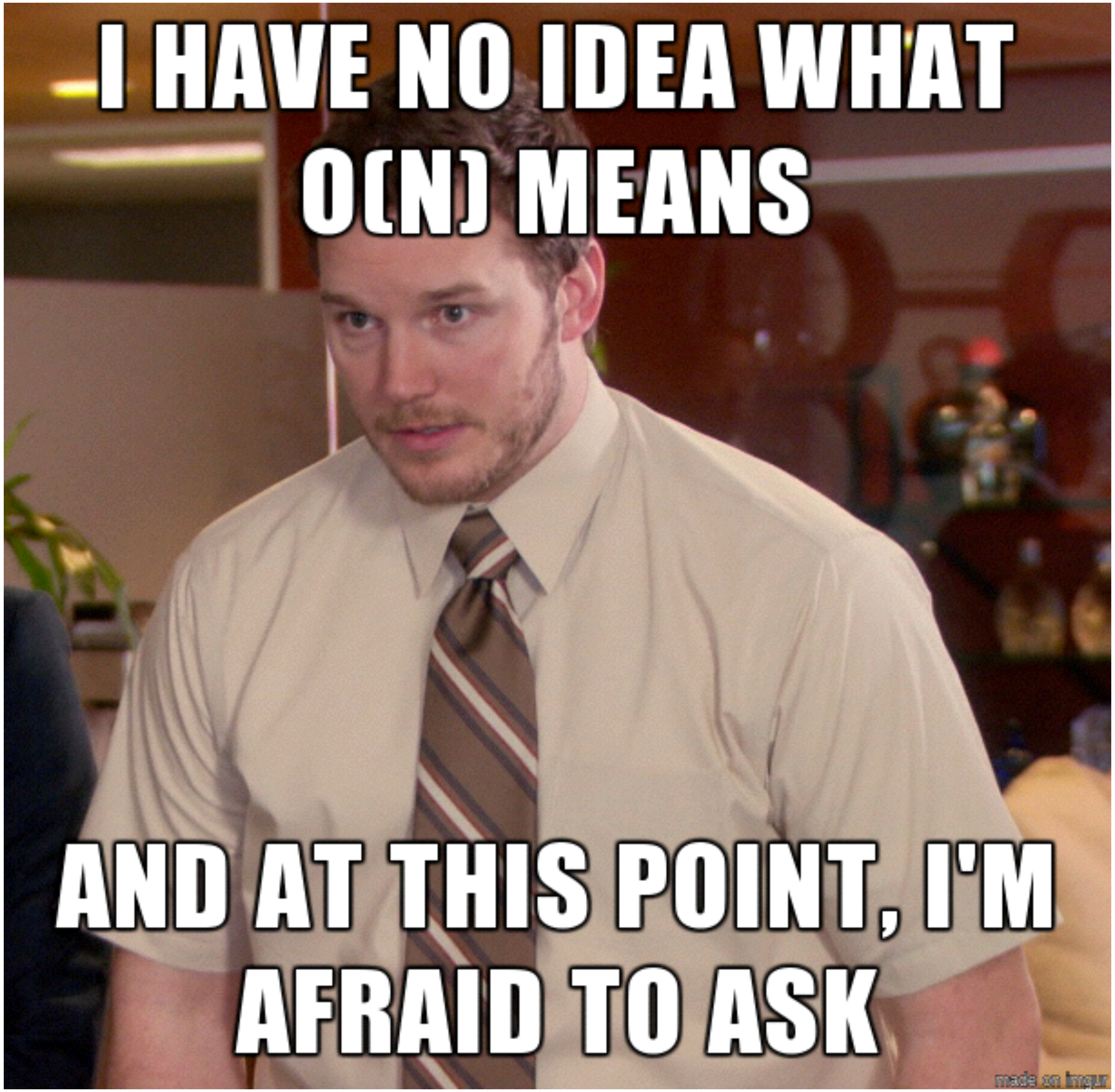
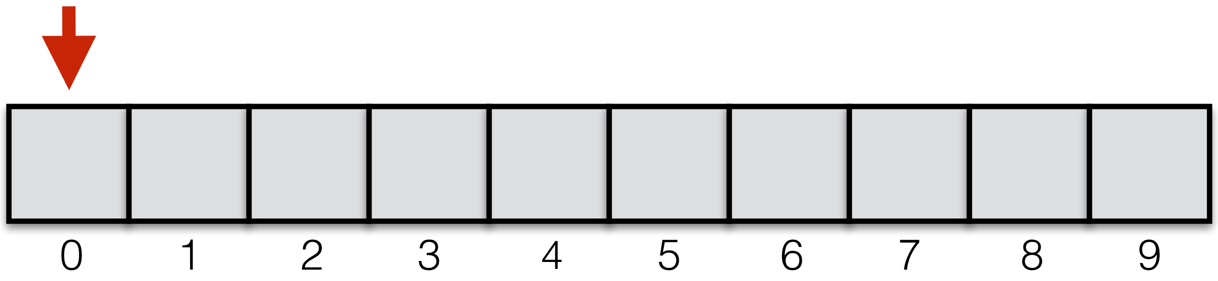


Photo Credit: CodeChef

Big O

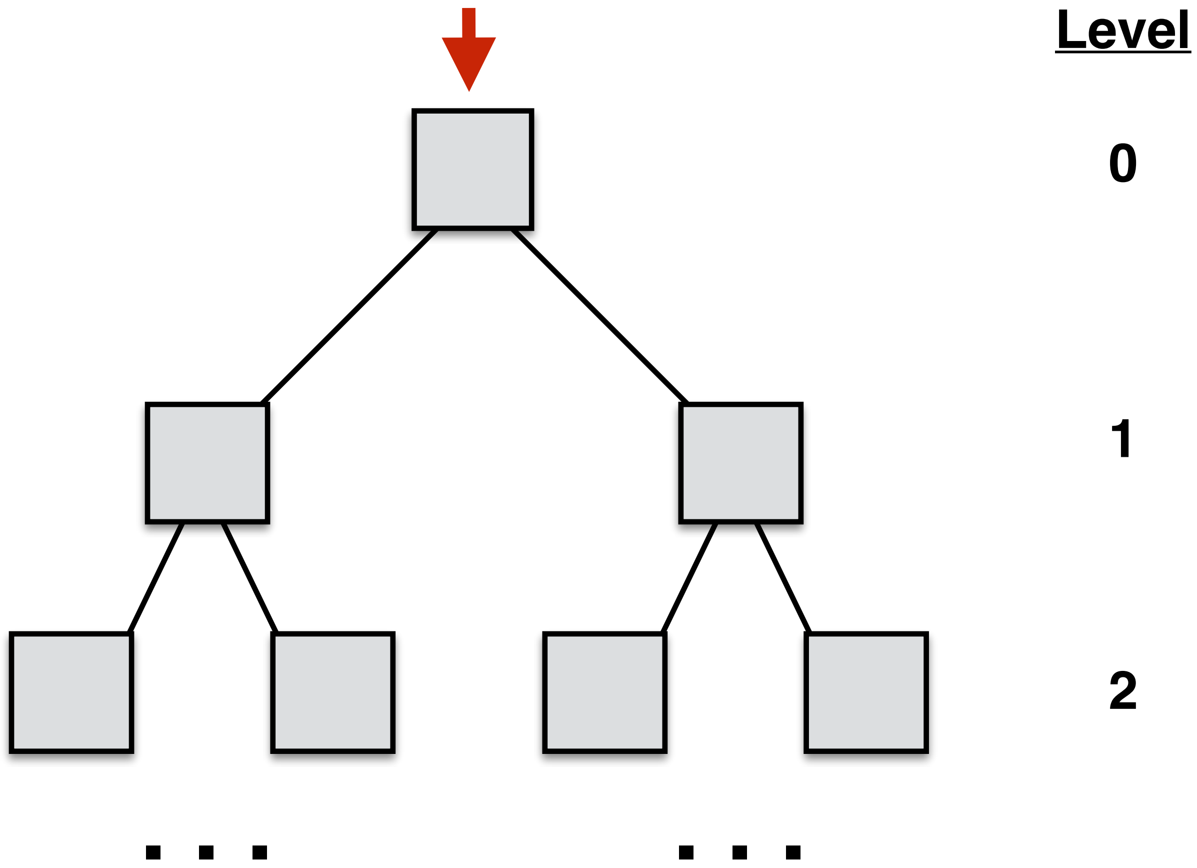
- Ex. Linear Search
  - running time “grows like” ***n***
  - at most some linear function of the input size ***n***
  - Ignoring the coefficients and low-order terms: **O(n)**



- *Read “O-notation” as “order."*
- *O(n) is thus read as "order n.”*
  - *a.k.a. "big-Oh of n”*
  - *a.k.a. "Oh of n.”*

Big O

- Ex. Binary Search
  - running time “grows like” ***log(n)***
  - at most some linear function of the input size ***n***
  - Ignoring the coefficients and low-order terms: **O(log(n))**



- *O(log(n)) is thus read as "order log n.”*
  - *a.k.a. "big-Oh of log n”*
  - *a.k.a. "Oh of log n."*

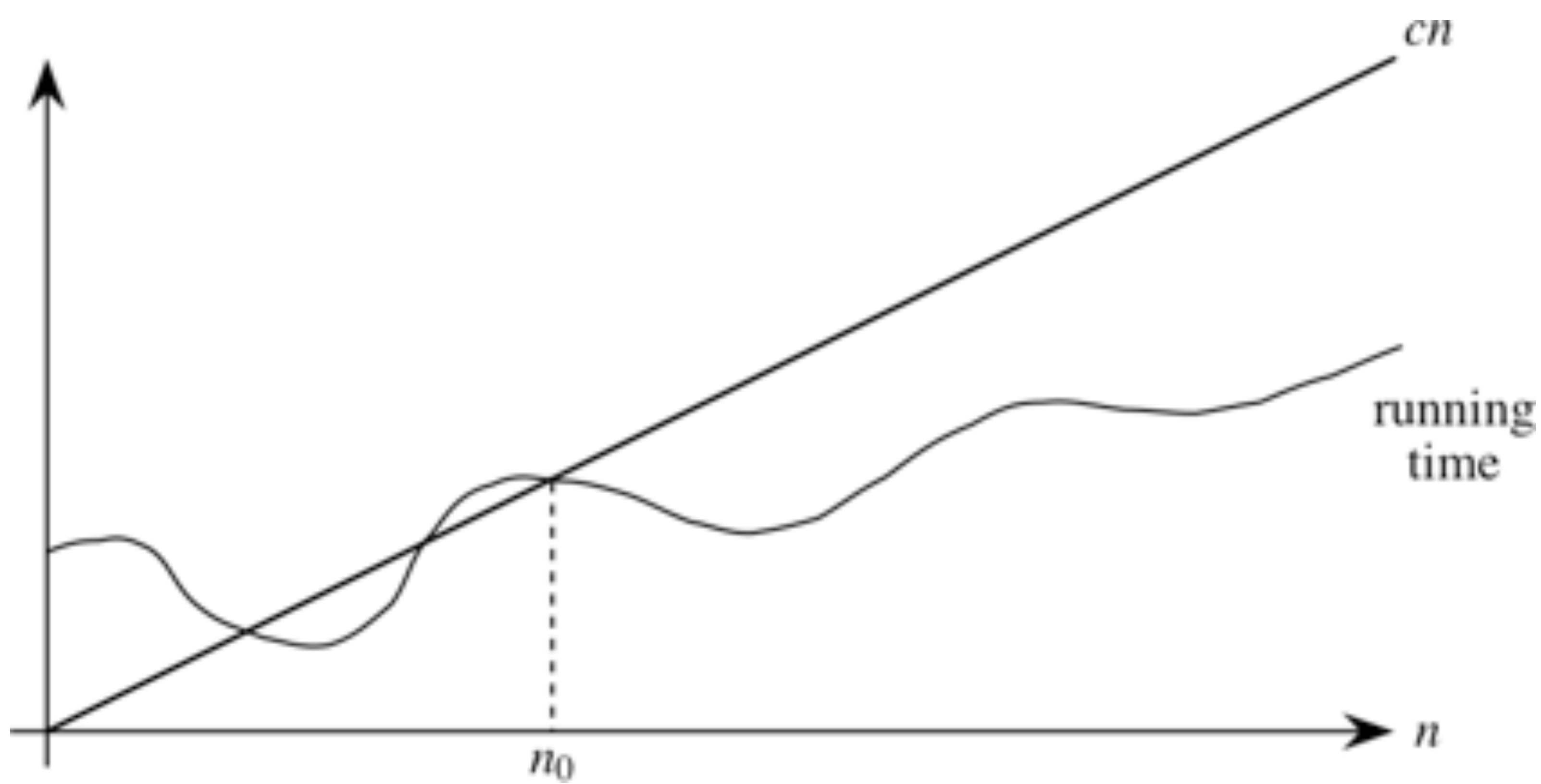
Why Big O?

- O-notation is used for what we call "asymptotic upper bounds.”
- By “asymptotic” we mean:
  - as the argument ***n*** gets large.
- By “upper bounds” we mean:
  - O-notation gives us a bound *from above* on how high the rate of growth is

Big O: Linear Upper-bound

Linear Case: O(n)

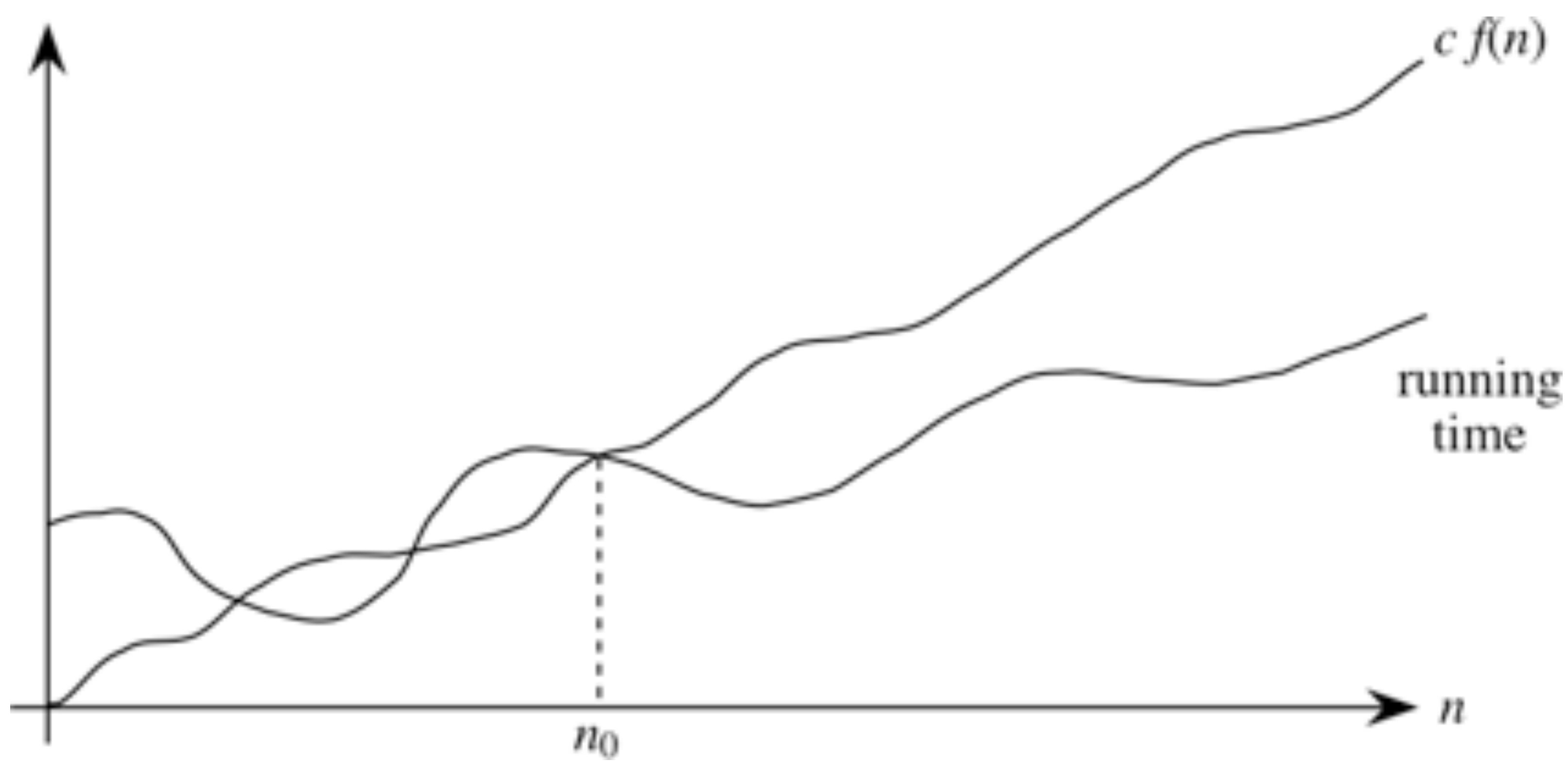
*A running time is O(n) if there exist positive constants n0 and c such that for all problem sizes n ≥ n0, the running time for a problem of size n is **at most** cn.*



Big O: General Upper-bound

General Case: O(f(n))

A running time is O(f(n)) if there exist positive constants n0 and c such that for all problem sizes n ≥ n0, the running time for a problem of size n is ***<u>at most</u>*** *c f(n).*



Big O: General Take-Aways

- *In general, when designing/choosing algorithms, we want as **slow** a rate of growth as possible, since if the running time grows slowly, that means that the algorithm is relatively fast for larger problem sizes.*

- *We usually focus on the **worst case** running time, for several reasons:*
  1. *Because computer scientists are very pessimistic…*
  2. *The worst case time gives us an **upper bound** on the time required for any input.*
  3. *It gives a **guarantee** that the algorithm **never** takes any longer.*
  4. *We don't need to make an educated guess and hope that the running time never gets much worse.*

Big Ω

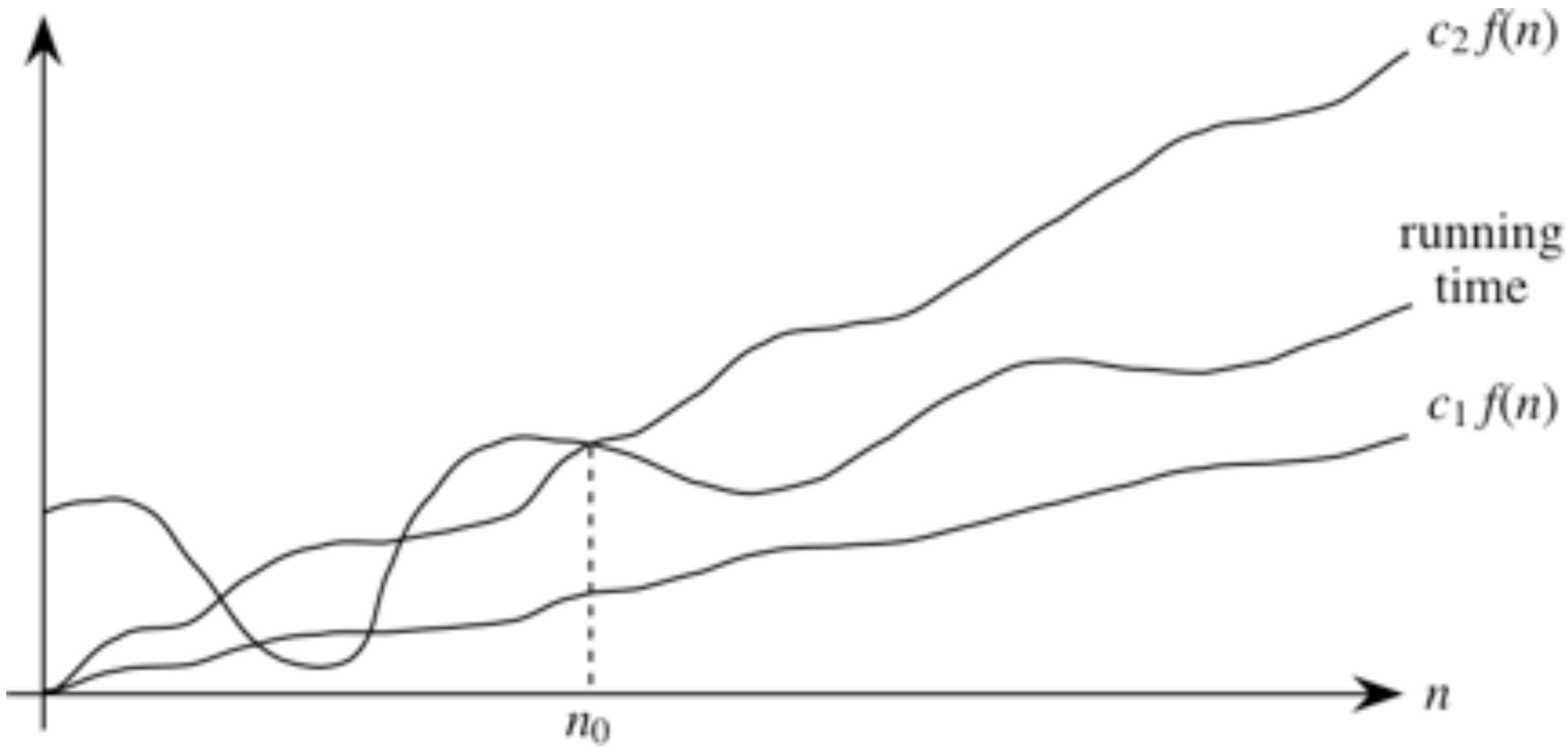
General Case: O(f(n))

A running time is Ω(f(n)) if there exist positive constants n0 and c such that for all problem sizes n ≥ n0, the running time for a problem of size n is ***<u>at least</u>*** *c f(n).*

Big Θ

General Case: O(f(n))

*A running time is Θ(f(n)) if there exist positive constants n0, c1, and c2 such that for all problem sizes n ≥ n0, the running time for a problem of size n is **at least** c1 f(n) and **at most** c2 f(n).*



Big O, Θ, and Ω: Recap

- Big O, Θ, and Ω — known as “asymptotic notations”
- “Asymptotic notations” provide ways to characterize the **rate of growth** of a function f(n).
- For our purposes, the function f(n) describes the running time of an algorithm.
- Asymptotic notation describes what happens as *n* gets **large**; we don't care about **small** values of *n*.

Working w/ Asymptotic Notation

While these definitions sometimes seem daunting, in practice there are some nice simplifications that we can use to make our lives easier

Constant factors don't matter…

Consider $f(n) = 1000n^2$.

I claim that this function is $\theta(n^2)$.

Since we can chose $c_1 = 1000$ and $c_2 = 1000$. Thus,

$c_1 n^2 \leq 1000n^2 \leq c_2 n^2$

$1000n^2 \leq 1000n^2 \leq 1000n^2$

Low-order terms don't matter, either…

Consider $f(n) = n^2 + 1000n$.

I claim that this function is $\theta(n^2)$.

If I choose $c_1 = 1$, then I have $n^2 + 1000n \geq c_1 n^2$, and so this side of the inequality is taken care of.

The other side is a bit tougher: Need to find a constant c_2 s.t. for sufficiently large n, I'll get that $n^2 + 1000n \leq c_2 n^2$.

Subtracting n^2 from both sides gives $1000n \leq c_2 n^2 - n^2 = (c_2 - 1)n^2$.

Dividing both sides by $(c_2 - 1)n$ gives $\frac{1000}{c_2 - 1} \leq n$.

Now, I pick $c_2 = 2$, so that the inequality becomes $\frac{1000}{2-1} \leq n$, or $1000 \leq n$.

Now I'm in good shape, because I have shown that if I choose $n_0 = 1000$ and $c_2 = 2$, then for all $n \geq n_0$, I have $1000 \leq n$, which we saw is equivalent to $n^2 + 1000n \leq c_2 n^2$.

Combining them (constants/low-order terms) still doesn't matter!

In combination, constant factors and low-order terms don't matter. If we consider a function like $1000n^2 - 200n$, we can ignore the low-order term $200n$ and the constant factor 1000, and therefore we can say that $1000n^2 - 200n$ is $\theta(n^2)$.