University of Virginia

# CS 2100: Data Structures & Algorithms 1

## Big-Oh Analysis

## {Orders of Growth}

Dr. Nada Basit // basit@virginia.edu

Spring 2022

# Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

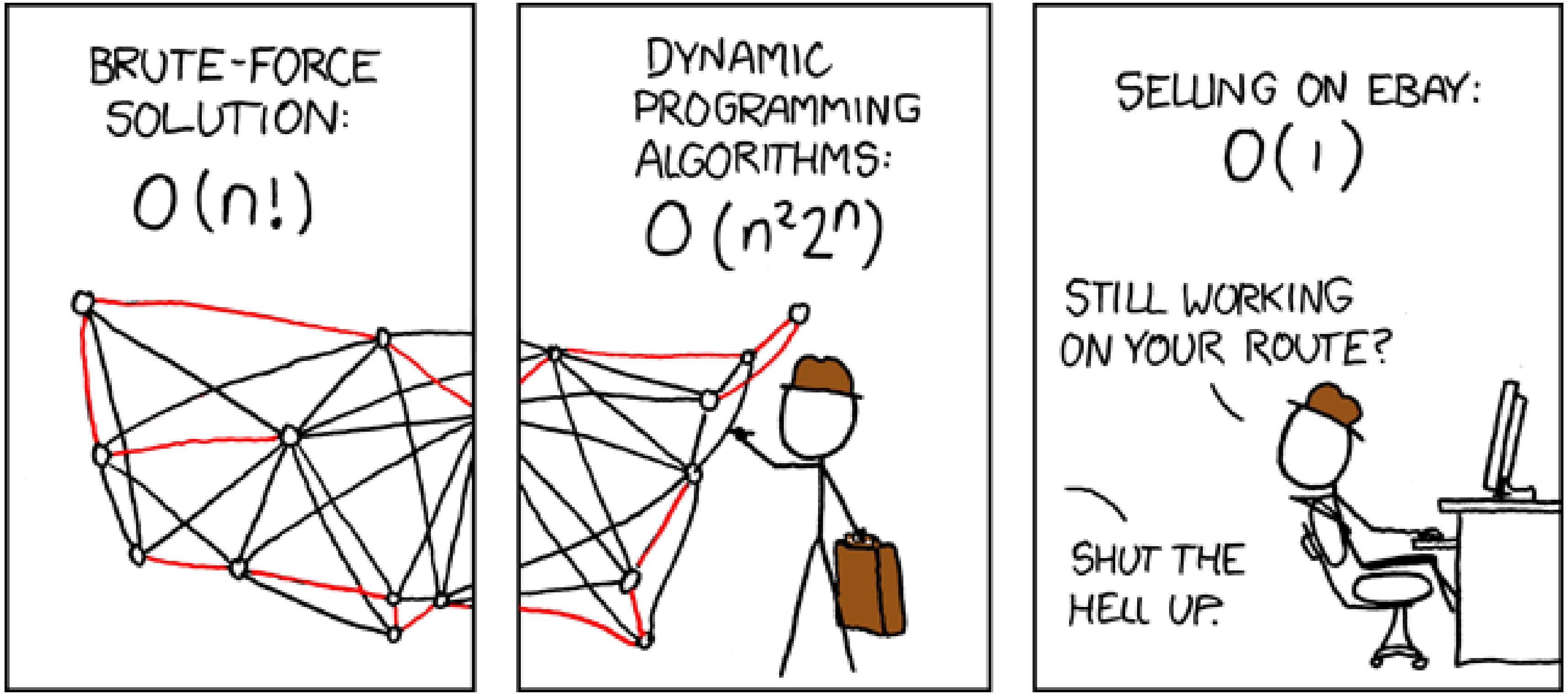
# Announcements

- **Lab tonight (Monday):**
  - Take a maximum of **two (2) quizzes** – ***on external testing site, linked to from Collab***
    - ONE (1) quiz from this week - **Vectors** (**30 minutes**)
    - ONE (1) quiz from:  Quiz 1, Quiz 2, or Quiz3  (retake; **optional**; **another 30 minutes**)
  - Show up to lab on-time!

- Prof. Floryan and I (along with other CS professors) are attending a **Computer Science Education Conference (ACM SIGCSE)** starting on Wednesday until Saturday.
  - Lectures for Wednesday and Friday will be recorded and posted for you to watch
    - **Wednesday**: continuation of today's lecture plus examples [Basit]
    - **Friday**: clarifying or finishing off material, plus Amortized analysis [Floryan]
  - Students from both sections will watch all posted videos for this week
  - Therefore, no class Wednesday or Friday of this week! – ***Have a great Spring Break!***

# Motivation/Goals

- Goal: Measure the *quality* of an **algorithm** or **method** in a data structure
  - E.g., is `find()` in LinkedLists faster than `find()` in an Array?
  - What about `get(index n)`?

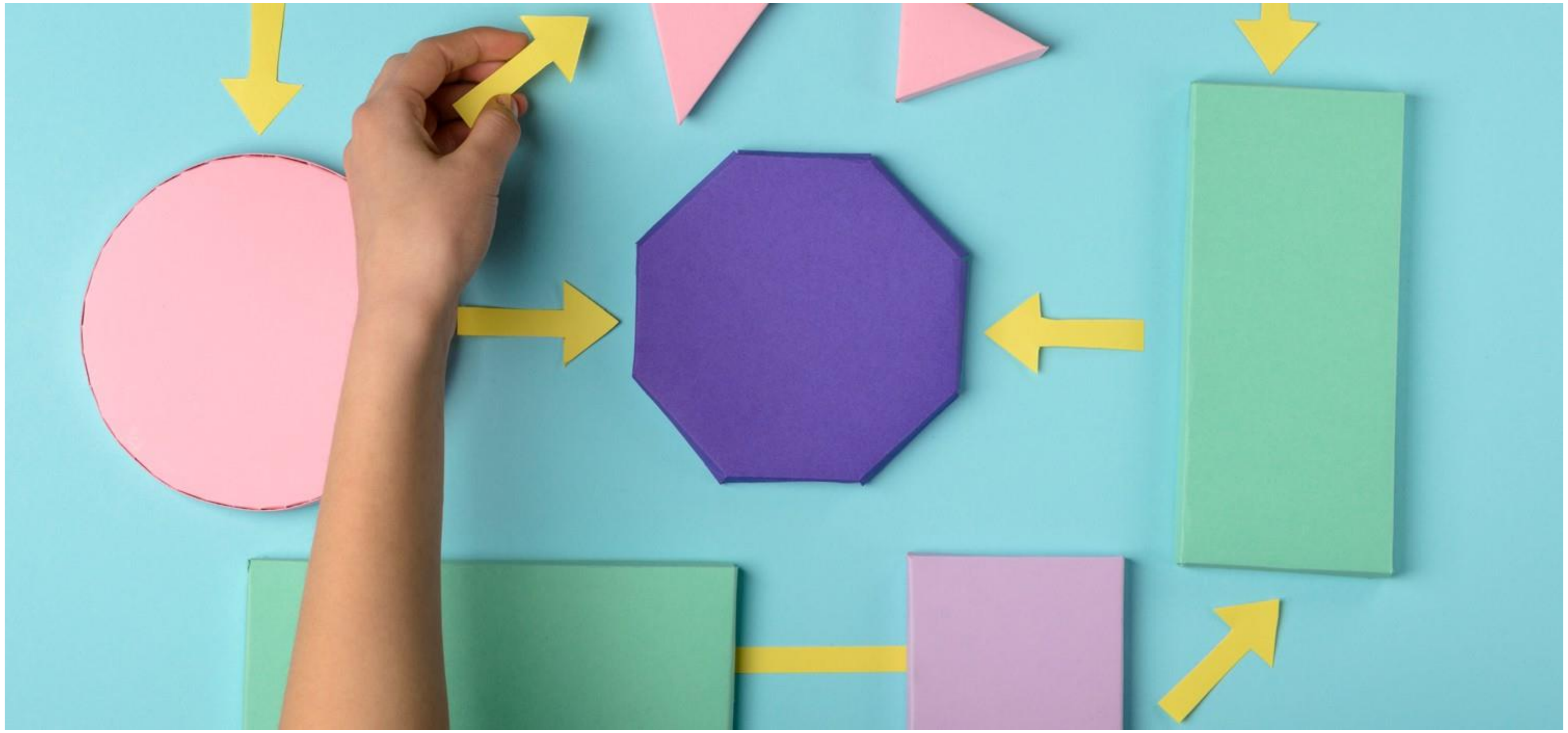


# Motivation/Goals

- Understand the nature of the *performance of algorithms*
- Understand *how we measure performance*
  - Used to describe *performance of various data structures*
- Begin to see the role of algorithms in the study of Computer Science

# Algorithms

- An ***algorithm*** is a detailed step-by-step method for solving a problem
- Computer algorithms, but other kinds too! [***Such as?***]



# Algorithms

- An ***algorithm*** is a detailed step-by-step method for solving a problem
- Computer algorithms, but other kinds too! [***Such as?***]

- *Properties of algorithms*
  - Steps are **precisely stated**
    - No ambiguity
    - Cannot be interpreted in more than one way
  - **Deterministic:** behaves the same way (based on inputs, previous steps)
  - **Terminates:** the <u>end</u> is clearly defined
  - Other properties: correctness, generality, efficiency

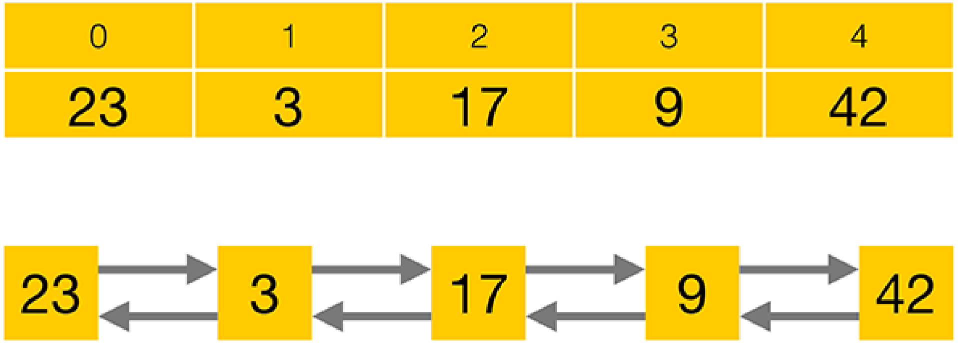
# Abstract Data Types

- **Data Structures**: A logical relationship among data elements designed to support specific data manipulation functions
  - Concrete: defined as an implementation
  - Examples: ArrayList, HashSet, trees, tables, stacks, queues

- **Abstract Data Types (ADT)**: a *model* of data items stored and a *set of operations* that manipulate the data model
  - Abstract: no implementation implied (**data abstraction**)
  - Examples: List, Set, … (think Java interfaces)
  - A particular data structure implements an ADT and defines *how* it is implemented

# ADTs

- ADTs define **operations** and a given data structure implements them
  - Think and design using ADTs, then **code** using data structures
  - There may be more than one data structure that implements the ADT we need – *so how do we decide?*
    - Compare the advantages and disadvantages
    - Efficiency / performance is often a major consideration
    - Ex: **ArrayList vs LinkedList**



# ADTs - How to compare the efficiencies of implementations?

- Example: ADT List can be implemented by an array, an ArrayList, a LinkedList, …
- Example: ADT Priority Queue can be implemented by an array, an ArrayList, or a Binary Heap (base structure is a binary *tree*)

- **So…** how do we compare efficiency of implementations?
- **Answer**: We compare the algorithms that implement the operations

- E.g.:
  the remove method of an **ArrayList**
  vs.
  the remove method of a **LinkedList**

# How Do We Compare the Efficiency of Implemented Algorithms?

Core question we would like to address today

*What* things are we comparing?

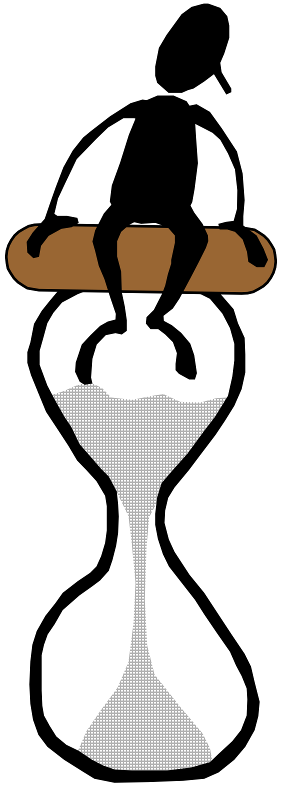
What do we mean by *efficiency*?

# Let's Define Efficiency…

- The **efficiency** of an algorithm measures the amount of resources consumed in solving a **problem of size *n***
  - CPU (time) usage, memory usage, disk usage, network usage, …
- In general, the resource that interests us the most is **time**
  - That is, how **fast** an algorithm can solve a problem of size ***n***
  - (We can use the same techniques to analyze the consumption of other resources, such as memory space)

# Why Not Just Time Algorithms?

*What do you think?*



*How about…*

**MacBook 2018** vs **4GHz Core i7** vs **Rasberry Pi** (ARM) vs **Apple M1** ???

Not a fair comparison!

(This is called **BENCHMARKING**, and is useful in certain circumstances.)

# Considerations…

Algorithm A
(implementing task "X")

Algorithm B
(implementing task "X")

- PC vs. MAC
- Python vs. Java
- Programmer 1 vs. Programmer 2
- Laptop vs. super computer
- MAC vs. MAC

---

- (Data) Input: 10 vs. Input: 100,000
- (Data) Input: 10,000 Sorted vs. Input: 10,000 Random

# Why Not Just Time Algorithms?

- We want a measure of work that gives us a direct measure of the *efficiency* of the algorithm without introducing differences in:

  - Computer hardware
  - Programming language
  - Programmer skills (in coding the algorithm)
  - (Other implementation details)
  - The *size of the input* – bits, # items in data structure, …
  - The *nature of the input*
    - Best-case, worst-case, average
    - *E.g. searching a sorted vs. a randomized list*

# Measuring Performance

- We need a way to formulate general guidelines that allow us to state that, *for any arbitrary input, one method is likely to perform better than the other*

- The time it takes to solve a problem is usually an increasing function of its **size** (n) – *the bigger the problem, the longer it takes to solve*
- We need a formula that associates ***n***, the problem size, with ***t***, the processing time required to obtain a solution
- This relationship can be expressed as: $\boxed{t = f(n)}$

# Analysis of Algorithms

- Analysis of Algorithms: use mathematics as our tool for analyzing algorithm performance
  - Measure the algorithm itself, *its nature*
  - Not its implementation or its execution

- We need something to **count**!
  - Cost or number of steps is a function of input size *n*:
    e.g. for input size *n*, cost is ***f(n)***
  - Count <u>all steps</u> in an algorithm? (*Hopefully avoid this!*)

# Some First Attempts

- **attempt 1:** Use a stopwatch and time each algorithm
  - Ok, but a bunch of confounding factors here (cpu speed, cache hits, other processes running, etc.)
  - High variability even with one algorithm / method

# Some First Attempts

- **attempt 2**: Count the number of lines of code / operations
  - Good! But some issues still
  - Some operations much faster than others, maybe not a big deal unless using them a lot
  - How do we account for loops? Really running those lines over and over.
  - If data structure / input is small, 10 operations versus 8 doesn't really matter

# Some First Attempts

- **attempt 3:** Count operations and measure how they scale up!!
  - Not perfect, but very useful theoretical measure of an operation's efficiency
  - Count the operations as a function of how big the input is.
  - Example: find() in a Linked List. How many operations? Depends on size of the list!

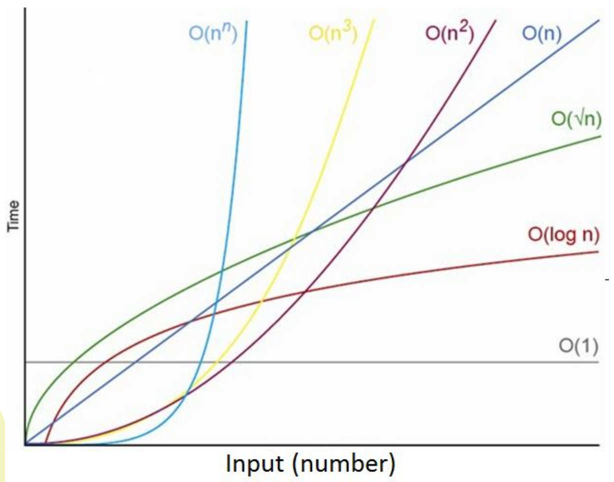
# Counting Operations

- **Strategy**:  choose one operation or section of code to count
  - Total work is always roughly proportional to how often that part is done

- So, we'll just count:
  - An algorithm's "basic operation," or
  - An algorithm's "**critical section**"

# Asymptotic Analysis (Characterizing the performance of an algorithm)

- **Algorithmic complexity** is concerned with how fast or slow a particular algorithm performs.
  - *How long will a program run on an input?*
  - *How much space will it take?*
  - *Is the problem solvable?*



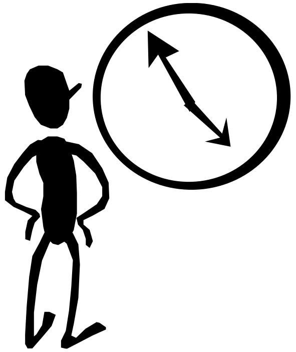

- An understanding of algorithmic complexity provides programmers with insight into the efficiency of their code

- **Asymptotic Analysis**: an estimate of time as a function of the input size, *n*, *as n gets large*
  - **As n gets large:** it's only when n becomes large that differences become apparent

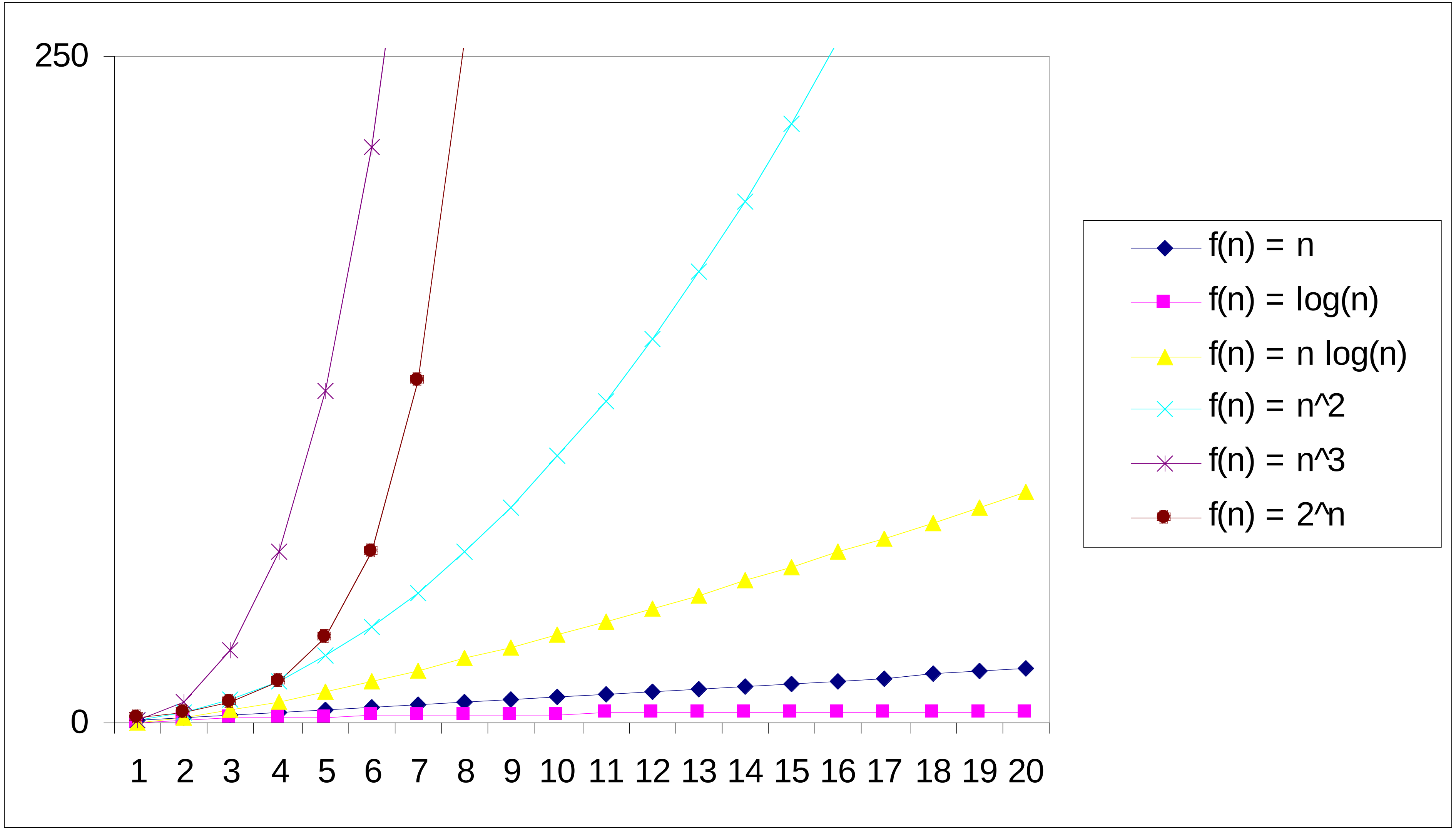
# Asymptotic Analysis

- **Asymptotic Analysis**: an estimate of time as a function of the input size, *n, as n gets large*
  - **As n gets large:** it's only when n becomes large that differences become apparent
- The **asymptotic behavior/growth rate** of a function $f(n)$ refers to the growth of $f(n)$ as ***n*** **gets large**
  - A mathematical concept (i.e. on its own has nothing to do with code / CS but we use it in CS). Describes the growth rate of something as a function
  - E.g., $\mathbf{n^2}$, quadratic, grows faster than **n**, linear.
  - "Asymptotic" – how do things change as the input size ***n*** gets larger? How **scalable** is the algorithm (how slow will it be on large inputs?)

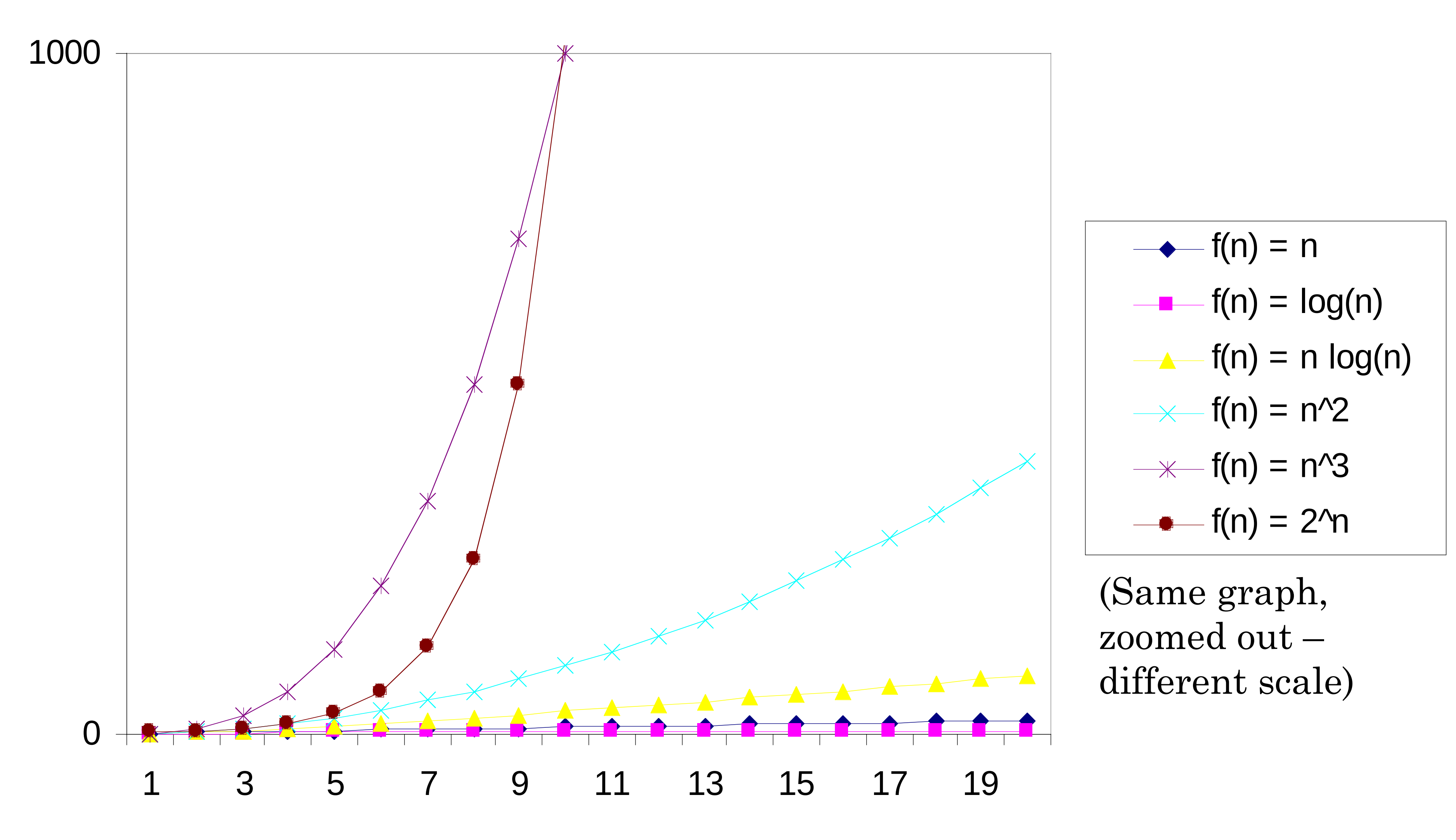
- <u>**Rule of thumb**</u>: the **slower** the asymptotic growth rate, the ***better*** the algorithm



# Comparison of Growth Rates



# Comparison of Growth Rates ("zoomed out")




(Same graph, zoomed out – different scale)

# Time Complexity

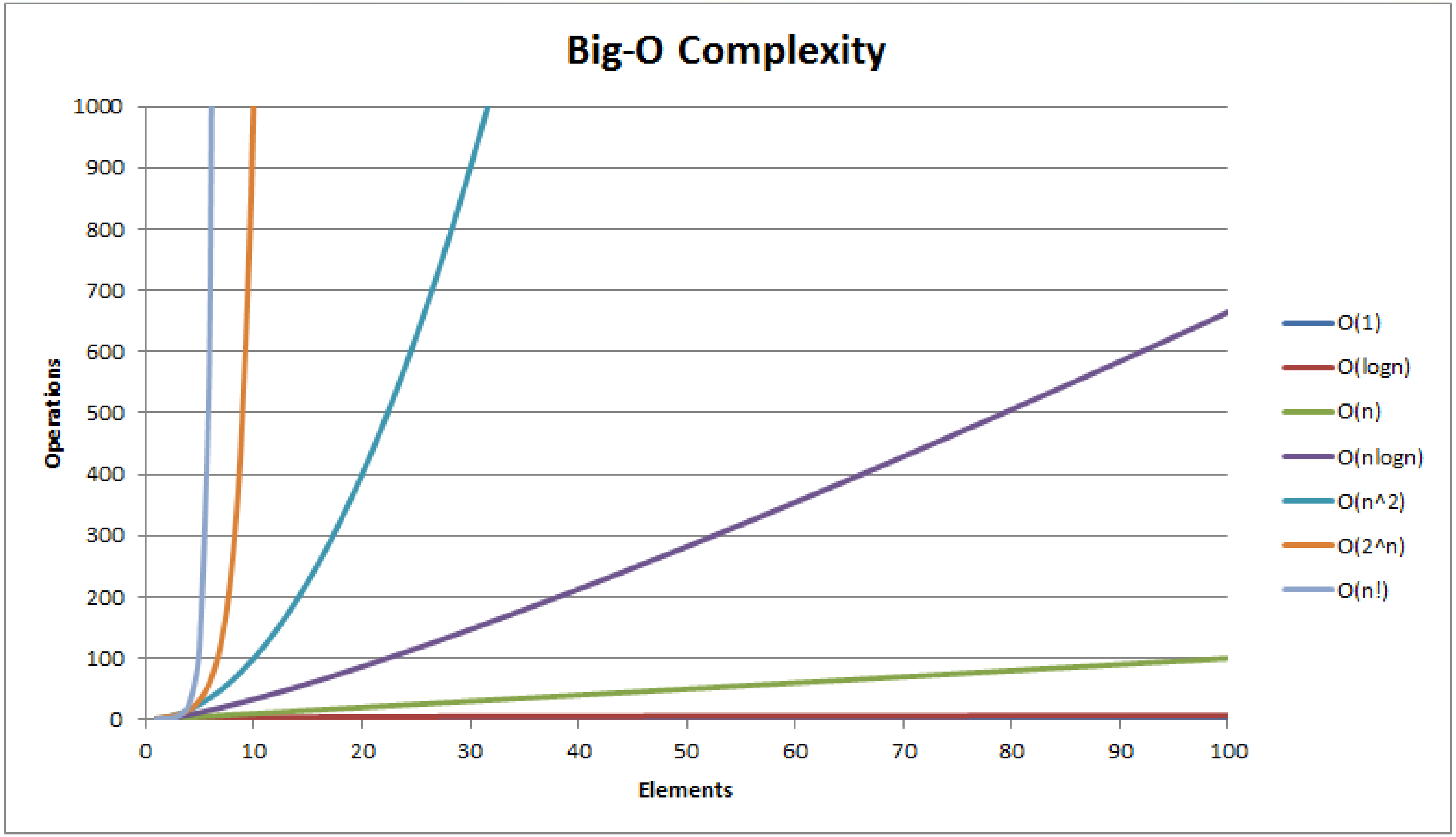
- *Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity*

- For large inputs, are these functions really different?
  - $f(n) = 100n^2 + 50n + 7$
  - $f(n) = 20n^2 + 7n + 2$

  They are both quadratic functions

- **Order class:** a “label” for all functions with the same *highest-order term*
  - **$O(n^2)$** : ***Big-Oh*** notation [typically used more often]
  - **$\Theta(n^2)$** : ***Big-Theta*** notation

# Time Complexity : Common order classes



# Classifying Functions by Their Asymptotic Growth Rates

- **Asymptotic growth rate** or **asymptotic order**
  - Comparing and classifying functions that ignores constant factors and small inputs.
- The sets are big-omega, big-theta, and big-oh:
  - **$\Omega(g)$:** functions that grow *at least as fast as* g
  - **$\Theta(g)$:** functions that grow *at the same rate as* g
  - **$O(g)$:** functions that grow *no faster than* g

# Why Do We Care?

- Some data structures are faster than others
  - Each data structure has some operations that are **fast**, and some that are **slow**
- We need a way to compare them
- This allows us to:
  - Better choose the data structures that we will use
  - Better design additional data structures

# Input Sizes

- Your algorithm does not matter if you have **10 elements** ***(small size)***
  - A **bogosort** will work just fine [http://en.wikipedia.org/wiki/Bogosort]
- Consider ***big input sizes***:
  - UVa's e-mail probably has about 100,000 e-mail addresses
  - OpenStreetMap, for driving routes, has over **3.2 billion nodes** and **5.1 million GPS points** (ref) (as of Feb 2016)

# Even for Smaller Input Sizes...

- All times are in ms (1/1000th of a second)

| Data Structure | Total time | Insert time | Search time | Delete time |
|---|---|---|---|---|
| Vector | 17,311 | 30 | 12,620 | 4,661 |
| ArrayList | 17,281 | 28 | 12,609 | 4,644 |
| LinkedList | 24,255 | 54 | 17,934 | 6,267 |
| HashSet | 122 | 103 | 9 | 10 |

# Assumptions

- We have measured the running time of our program with different input sizes, and that result is encapsulated in some **function *f(n)***
  - ***n*** is the input size, and is always **a positive integer**

- We have some other function ***g*** that we want to compare our program to

- So, we will compare ***f(n)*** to ***g(n)***, such as:
  - $f(n) \in O(g(n))$
  - $f(n) \notin \Omega(g(n))$

# Worst-Case Scenario

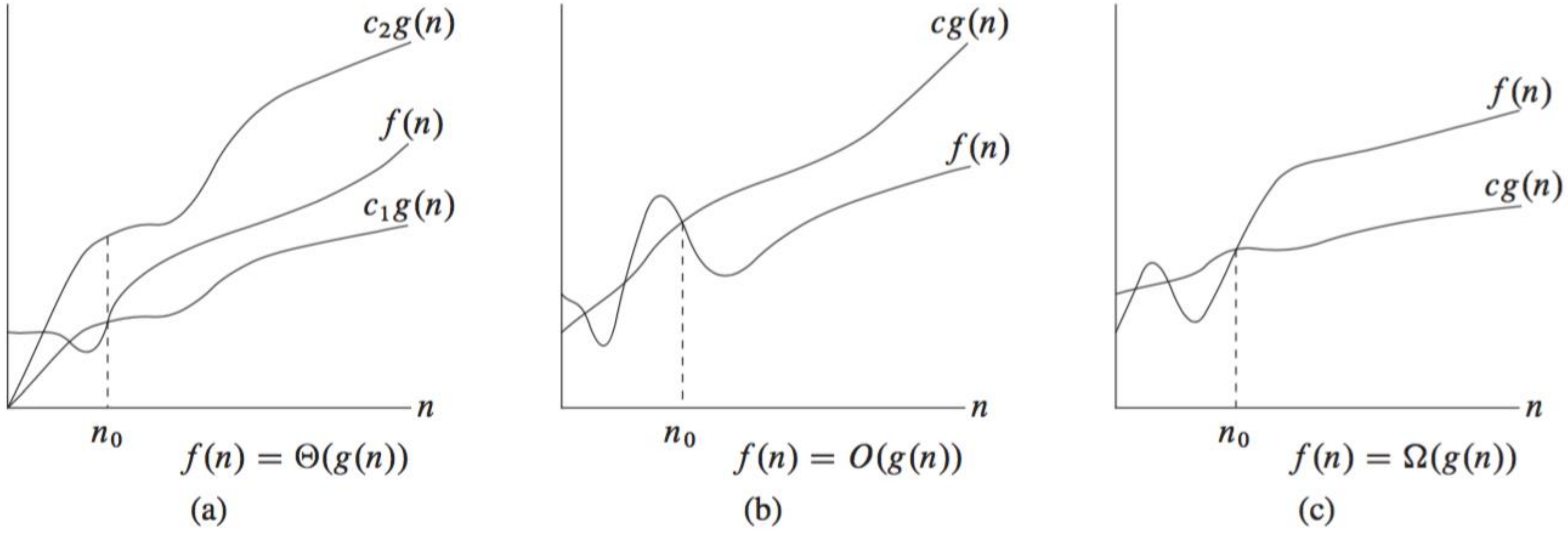
- We always analyze the **worst-case** run-time
  - It makes no sense to analyze the best case, as that is rarely likely to happen
  - And the average case (if you could even define what that is) may not be representative and is not used in these analyses either
- It is often not until a worst-case scenario happens that ‘bad’ / ‘incorrect’ things happen
  - So, we want to pay attention to how ‘bad’ our algorithms function at these times
- A more formal definition of worst case, should you be interested, can be found here
  - ( http://en.wikipedia.org/wiki/Worst_case )

# The Sets O(G), Θ(G), Ω(G)

- Let f and g be functions from the non-negative integers into the positive real numbers
- For some real constant **c** > 0 and some non-negative integer constant $\boldsymbol{n_0}$
  - O(*g*) is the set of functions *f*, such that:
    - $f(n) \leq \mathbf{c} * g(n)$ for all $n \geq \boldsymbol{n_0}$ [*no faster than* g; an ***asymptotic*** upper bound]
  - Ω(*g*) is the set of functions *f*, such that:
    - $f(n) \geq \mathbf{c} * g(n)$ for all $n \geq \boldsymbol{n_0}$ [*at least as fast as* g; an ***asymptotic*** lower bound]
  - $\Theta(g) = O(g) \cap \Omega(g)$
    - Θ(*g*) is the asymptotic order of g [*at the same rate as* g; an ***asymptotic*** tight bound]
    - or the *order* of g

# Asymptotic Bounds

- For the sets big-oh O(g), big-theta Θ(g), and big-omega Ω(g), remember these meanings:
  - O(*g*): functions that grow *no faster than* g; an ***asymptotic*** upper bound [figure (b)]
  - Ω(*g*): functions that grow *at least as fast as* g; an ***asymptotic*** lower bound [figure (c)]
  - Θ(*g*): functions that grow *at the same rate as* g; an ***asymptotic*** tight bound [figure (a)]



# Big-Oh Examples

- $f(n) \in O(g(n))$ means that there are positive constant **c** and some non-negative integer constant $n_0$ such that $f(n) \le c * g(n)$ for all $n \ge n_0$

- Is $n \in O(n^2)$?
  - Yes $c = 1$, $n_0 = 2$ works fine
- Is $10n \in O(n)$?
  - Yes $c = 11$, $n_0 = 2$ works fine
- Is $n^2 \in O(n)$?
  - **No!** ***No matter what values for c and $n_0$ we pick, $n^2 > c * n$ for big enough n***

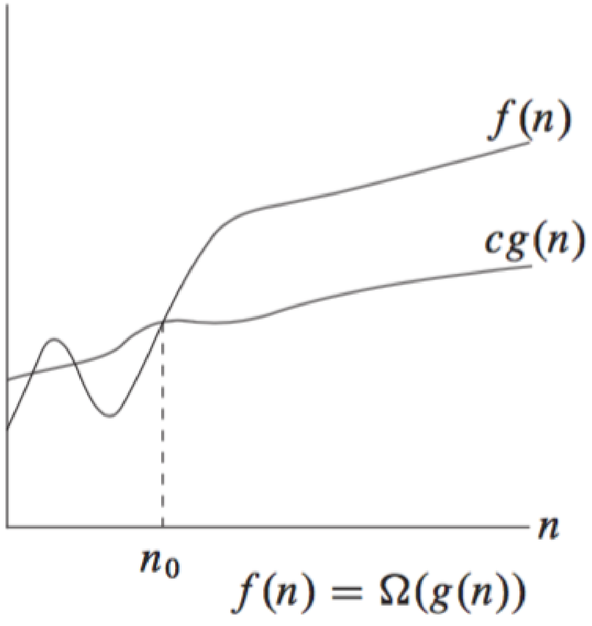
Greater than, not $\le$

# Given $F \in O(H)$ and $G \notin O(H)$, Which of These are True?

1. For ***all*** positive integers $m$, $f(m) < g(m)$.
2. For ***some*** positive integer $m$, $f(m) < g(m)$.
3. For ***some*** positive integer $m_0$, and ***all positive*** integers $m > m_0$, $f(m) < g(m)$.
4. 1 and 2
5. 2 and 3
6. 1 and 3

# Lower Bound: Ω (Omega)

- $f(n) \in \Omega(g(n))$ means:
  - There are positive constants **c** and $n_0$ such that $f(n) \geq c * g(n)$ for all $n \geq n_0$
  - The difference from big-oh is the ≥ in big-omega versus ≤ in big-oh
- This is a *lower* bound



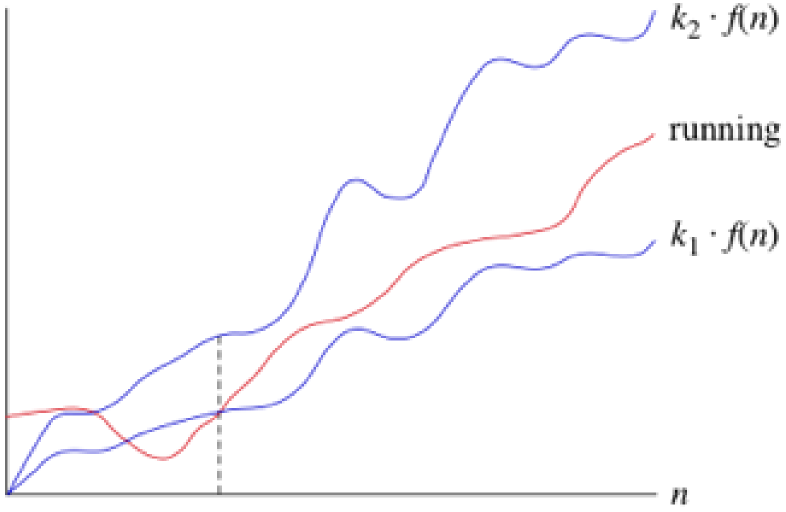
# Θ Theta ("Order Of")

- Intuition: the set $\Theta(f)$ is the set of functions that grow ***as fast as f***
- Definition: $f(n) \in \Theta(g(n))$ if and only if **both**:
  1. $f(n) \in O(g(n))$ -- *upper* bound
  2. $f(n) \in \Omega(g(n))$ -- *lower* bound

- Note that we do not have to pick the same **c** and $n_0$ values for cases 1 and 2
- When we say, *"f is order g"* that means $f(n) \in \Theta(g(n))$

# Running time that is Θ(f(n)) for some function f(n)

We are not restricted to just $n$ in big-Θ notation. We can use any function, such as $n^2$, $n \log_2 n$, or any other function of $n$. Here's how to think of a running time that is $\Theta(f(n))$ for some function $f(n)$:




Bounded both above and below:
***asymptotic*** tight bound

Once $n$ gets large enough, the running time is between $k_1 \cdot f(n)$ and $k_2 \cdot f(n)$.

Image from KhanAcademy.org