UNIVERSITY of VIRGINIA

# CS 2100: Data Structures & Algorithms 1

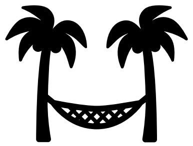
## Big-Oh Analysis (Pt. 2)
## {Growth Rates and Examples}

Dr. Nada Basit // basit@virginia.edu

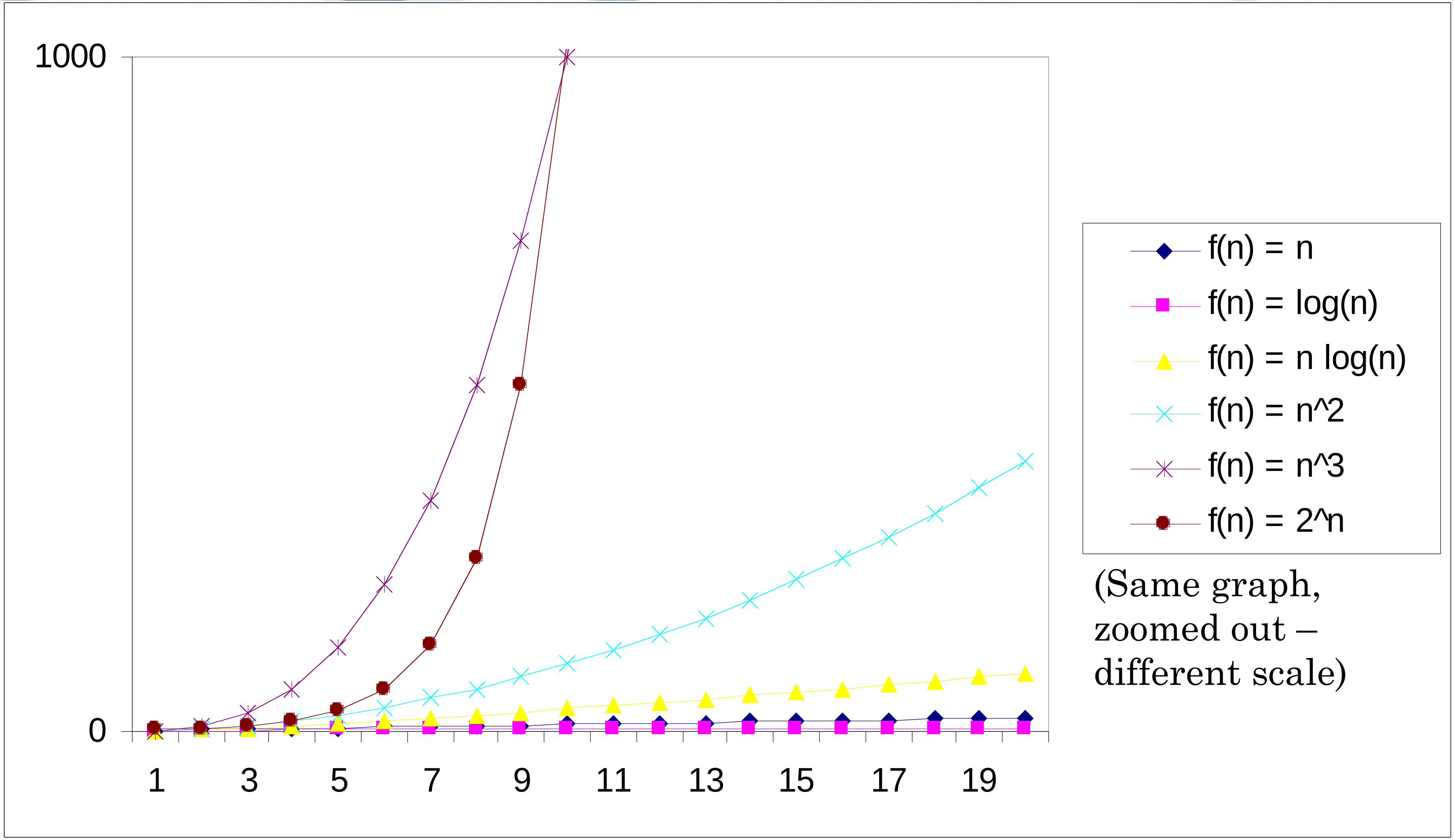
Spring 2022

# Announcements

- Prof. Floryan and I (along with other CS professors) are attending the **53rd ACM Technical Symposium on Computer Science Education**
  - This conference starts on Wednesday, March 2 and ends on Saturday, March 5, 2022.
- Therefore, lectures for Wednesday (today) and Friday will be recorded
  - **Wednesday**: continuation of Monday’s lecture plus examples [Basit]
  - **Friday**: finishing off and clarifying Asymptotic Complexity material, plus Amortized analysis [Floryan]
- Students from *both sections* will watch **all** posted videos for this week (by Basit and Floryan)
- Therefore, no class Wednesday or Friday of this week!
- Next week is Spring Break ☺ – ***Have a great Spring Break!***



# Comparison of Growth Rates ("zoomed out")




(Same graph, zoomed out – different scale)

# Order Class Details - Summary
## [Big-Oh; Big-Theta; Big-Omega]

$O(n^2)$: **upper-bound** on how inefficient an algorithm can be
- Set of all functions that are at most $n^2$
- Functions that grow the same rate or slower than $n^2$
- As efficient as $n^2$, but no worse

$\Theta(n^2)$: **tight-bound** on how inefficient an algorithm can be
- Set of all functions that are exactly $n^2$
- As efficient as $n^2$, no worse and no better

$\Omega(n^2)$: **lower-bound** on how inefficient an algorithm can be
- Set of all functions that are at least $n^2$
- As efficient as $n^2$, but no better

# Order Class Details - Summary [Big-Oh; Big-Theta; Big-Omega]

$O(n^2)$: **upper-bound** on how inefficient an algorithm can be

- Set of all functions that are at most $n^2$
- Functions that grow the same rate or slower than $n^2$
- As efficient as $n^2$, but no worse

$\Theta(n^2)$: **tight-bound** on how inefficient an algorithm can be

- Set of all functions that are exactly $n^2$
- As efficient as $n^2$, no worse and no better

$\Omega(n^2)$: **lower-bound** on how inefficient an algorithm can be

- Set of all functions that are at least $n^2$
- As efficient as $n^2$, but no better

# [*Reminder*] 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?
  - $f(n) = 100\mathbf{n}^2 + 50\mathbf{n} + 7$
  - $f(n) = 20\mathbf{n}^2 + 7\mathbf{n} + 2$

  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

# Highest-order Term

- If a function that describes the growth of an algorithm has several terms, its order of growth is determined by the fastest growing term.
- Smaller terms have some significance for small amounts of data. Constants are also eventually *ignored*
- As ***n*** gets large, the highest order term will dominate (*asymptotically*)

$$f(n) = \not{100}\mathbf{n^2} + \not{50}\mathbf{n} + \not{7} \rightarrow \text{Simply } O(\mathbf{n^2})$$

# Common Order Classes / Growth Rates

- Order classes group "equivalently" efficient algorithms
  - O(1) – constant time! Input size doesn't matter
  - O(lg n) – logarithmic time. Very efficient. E.g. binary search (after sorting)
  - O(n) – linear time E.g. linear search
  - O(n lg n) – log-linear time. E.g. best sorting algorithms
  - $O(n^2)$ – quadratic time. E.g. poorer sorting algorithms
  - $O(n^3)$ – cubic time. E.g. matrix multiply
  - ….
  - $O(2^n)$ – exponential time. Many important problems, often about optimization
  - O(n!) – factorial time. E.g. all permutations of a string

# Common Growth Rates

| Function | Name |
|---|---|
| $c$ | constant |
| $\log n$ | logarithmic |
| $\log^2 n$ | log-squared |
| $n$ | linear |
| $n \log n$ | log-linear |
| $n^2$ | quadratic |
| $n^3$ | cubic |
| $2^n$ | exponential |

# A Note About Logs

- The difference between $\log_{10}(n)$ and $\log_2(n)$ is always a **constant**
  - Specifically, about 3.322
  - Since we don't care about constants in these analyses, we'll *ignore* the log base
- Most things in computer science are log base 2 anyway...

# Time Complexity: Order Classes Details

- What does the label mean? $O(n^2)$
  - Set of all functions that grow at the same rate as $n^2$ **or** more slowly
  - i.e. as efficient as any "$n^2$" or more efficient, but no worse
  - An **upper-bound** on how inefficient an algorithm can be
- Usage: We might say: Algorithm A is $\mathbf{O(n^2)}$
  - Means: Algorithm A's efficiency grows like a quadratic algorithm **or** grows more slowly (as *good or better*)
- What about that other label, e.g., $\mathbf{\Theta(n^2)}$?
  - Set of all functions that grow at **exactly** the same rate
  - *A more precise bound – as efficient as $n^2$, no worse and no better*

# Does Order Class Matter?

- For **small** inputs…
  - No
- For many real problems (with usually **large** inputs)…
  - Yes!

# Big-O is a Good Estimate

- **For large values of N,** Big-O is a good approximation for the running time of a particular algorithm. The table below shows the observed times and the estimated times

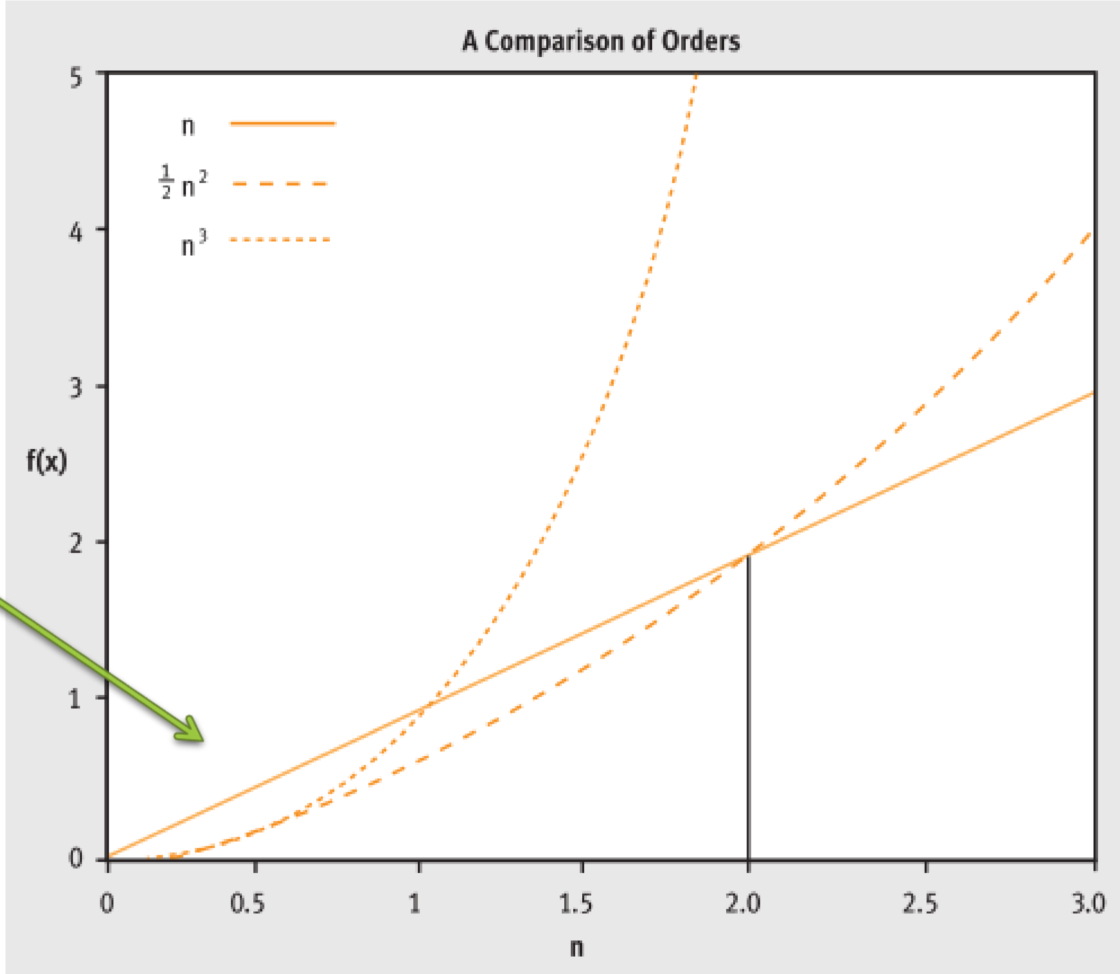
| *N* | Observed time | Estimated time | Error |
|---|---|---|---|
| 10 | 0.12 msec | 0.09 msec | 23% |
| 20 | 0.39 msec | 0.35 msec | 10% |
| 40 | 1.46 msec | 1.37 msec | 6% |
| 100 | 8.72 msec | 8.43 msec | 3% |
| 200 | 33.33 msec | 33.57 msec | 1% |
| 400 | 135.42 msec | 133.93 msec | 1% |
| 1000 | 841.67 msec | 835.84 msec | 1% |
| 2000 | 3.35 sec | 3.34 sec | < 1% |
| 4000 | 13.42 sec | 13.36 sec | < 1% |
| 10,000 | 83.90 sec | 83.50 sec | < 1% |

# Asymptotically Superior Algorithm

- If we choose an **asymptotically superior algorithm** to solve a problem, we will not know exactly how much time is required, but we know that **as the problem size increases** there will always be a point beyond which the lower-order method takes less time than the higher-order algorithm
- Once the problem size becomes sufficiently large, the asymptotically superior algorithm always executes more quickly
- The next figure demonstrates this behavior for algorithms of order O(n), $O(n^2)$, and $O(n^3)$

# Asymptotically Superior Algorithm

- For small problems, the choice of algorithms is not critical – in fact, the O($n^2$) or O($n^3$) may even be superior!
- However, as n grows large (larger than **2.0** in this case) the **O(n)** algorithm always has a superior running time and ***improves as n increases***




MSD Figure 5.1: Graphical comparison of complexity measures *O*(*n*), *O*(*n*2), and *O*(*n*3)

# Summary

We need an accurate way to measure **algorithm efficiency**, which is independent of hardware or external to the program factors

**Time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the **size of the input**

**Asymptotic analysis** (e.g.,using *Big-O*) is the fundamental technique for describing the efficiency properties of algorithms (*time complexity*)

**Big-O notation** describes the **asymptotic behavior** (upper bound) of algorithms on **large problems**

It is the fundamental technique for describing the efficiency properties of algorithms

# Summary : Common Complexity Classes

- O(1) – constant time
- O(lg n) – logarithmic time
- O(n) – linear time
- O(n lg n) – log-linear time
- $O(n^2)$ – quadratic time
- $O(n^3)$ – cubic time
- ….
- $O(2^n)$ – exponential time

***Increasing Complexity***

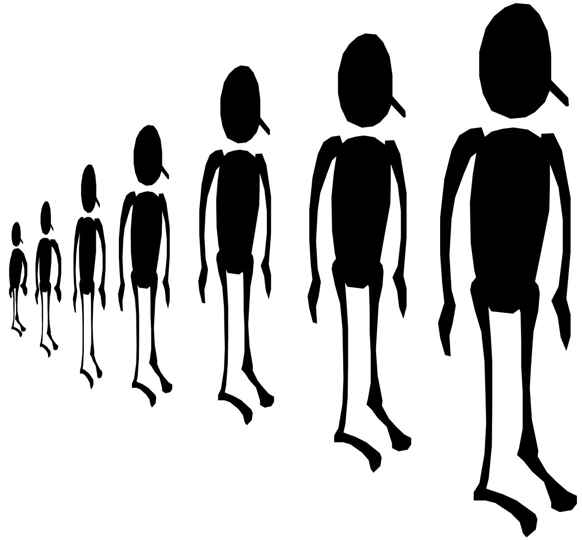
# Examples of Algorithms in Various Complexity Classes

# O(1) – Constant time

- The algorithm requires a fixed number of steps regardless of the size of the task (input)

**Examples**

- Push and Pop operations for a stack data structure (size n)
- Insert and Remove operations for a queue
- Conditional statement for a **loop**
- Variable declarations
- Assignment statements



# O(log n) – Logarithmic time

- Operations involving dividing the search space in ***half*** each time (taking a list of items, cutting it in half repeatedly until there's only one item left)

**Examples**

- Binary search of a sorted list of n elements
- Insert and Find operations for binary search tree (BST) with n nodes

# O(n) – Linear time

- The number of steps increase in proportion to the size of the task (input)

**Examples**

- Traversal of a list or an array… (size n)
- Sequential search in an unsorted list of elements (size n)
- Finding the max or min element in a list

# O(n lg n) – Log-linear time

- Typically describing the  behavior of more advanced sorting algorithms

**Examples**

- **Some of the best sorting algorithms**
  - Quicksort
  - Mergesort

# O($n^2$) – Quadratic time

- For a task of size 10, the number of operations will be 100
- For a task of size 100, the number of operations will be 100x100 and so on…

**Examples**

- Some poorer sorting algorithms
  - Selection sort of n elements, Insertion sort…
- Finding duplicates in an unsorted list of size n

***Think: doubly nested loops***

# $O(a^n)$ (a>1) – Exponential time

- Many interesting problems fall into this category…$O(2^n)$

**Examples**

- Recursive Fibonacci implementation
- Towers of Hanoi
- Generating all permutations of n symbols
- … many more!

# Calculating Running Time

So, what do we look for in the code to help us “count” and calculate running time?

Here are some general rules and tips with some additional examples

# General Rules for Running Time Calculations (Big-Oh)

- For loops
  - The number of times the for loop runs times the total running time of the statements inside the for loop
- Nested loops
  - Analyze from inside to out
    - Runtime of the statement * product of the sizes of the loops
- Consecutive statements
  - Additive (remember that we **remove** constants!)
- if/else
  - Time for the test plus the longer of the runtimes

# How to Tell The Running Time...

- You need to imagine how long the algorithm would run given varying input sizes
  - We will assume our input size is ***n***
- You should ask yourself...
  - How long will this algorithm take when ***n*** = 1?
  - When ***n*** = 100?
  - When ***n*** = 1,000,000,000?
- Recall that we always analyze the **worst case!**

# Running Time: Linear

- Linear (Θ(n)) running time
- Means that we have to process each element, and there is one step (or a constant number of steps) for each one
- Examples:
  - Printing (or otherwise iterating through) a list
  - Finding an element in an unsorted array or vector
  - Finding an element in a sorted or unsorted linked list
  - Doubling the size of a vector's underlying array

# Running Time: Log-Linear

- Log-linear ($\Theta(n \log n)$) running time
- Typically occurs when you are going to take a linear number of steps, but each one takes log n time
  - Or log n steps, each one of which takes n time
- Examples:
  - Fast sorts: merge sort and heap sort
  - Quicksort, on a good day *(but this is not guaranteed!)*
  - Inserting n elements into a data structure where each insert takes log n time

# Running Time: Quadratic

- Quadratic ($\Theta(n^2)$) running time
- Occurs when, for each input, you have to search through the entire input again
  - (among other ways it can occur)
- Examples:
  - Slow sorts: insertion sort, bubble sort, selection sort
  - Quicksort, *on a bad day*
  - Doubly nested for loops (where each loop goes 1 to ***n***)

# Running Time: Exponential

- Exponential ($\Theta(2^n)$) running time
- Typically means you are going to try every possible solution, and there are $2^n$ of them
- Examples (we have not seen any yet)
  - Trying to crack a binary combination lock by trying every single possibility
  - Traveling salesperson problem (we'll see that later this semester)
  - Satisfiability of a Boolean expression