Department of Computer Science

Intro to Asymptotic Analysis

or, Figuring out the basic complexity of a problem's solution

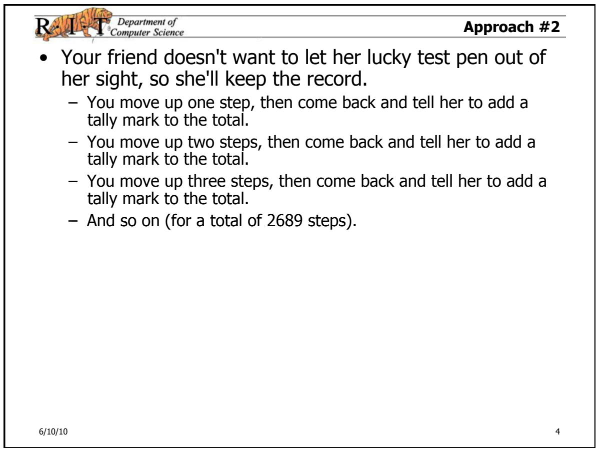
or, Trying to figure out which algorithm is "best"

Example problem

- Imagine that your friend wants to know how many steps there are in the Eiffel Tower. How can you count them?
- Several possible approaches (algorithms) come to mind....

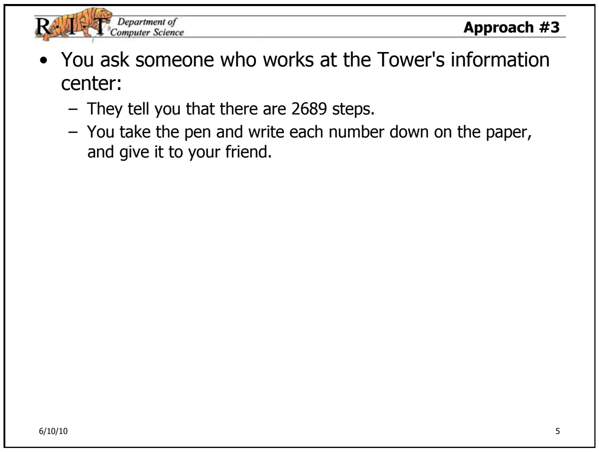
Approach #1

- You keep a record of each step:
  - You take a piece of paper and pen from your friend.
  - You then walk up the steps, making a tally mark on the paper for each one as you go.
  - When you get to the top, you walk back down again, and tell your friend, "There are 2689 steps."



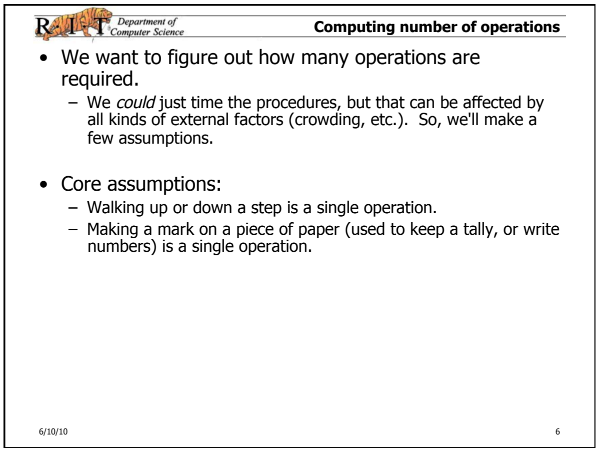
Approach #2

- Your friend doesn't want to let her lucky test pen out of her sight, so she'll keep the record.
  - You move up one step, then come back and tell her to add a tally mark to the total.
  - You move up two steps, then come back and tell her to add a tally mark to the total.
  - You move up three steps, then come back and tell her to add a tally mark to the total.
  - And so on (for a total of 2689 steps).



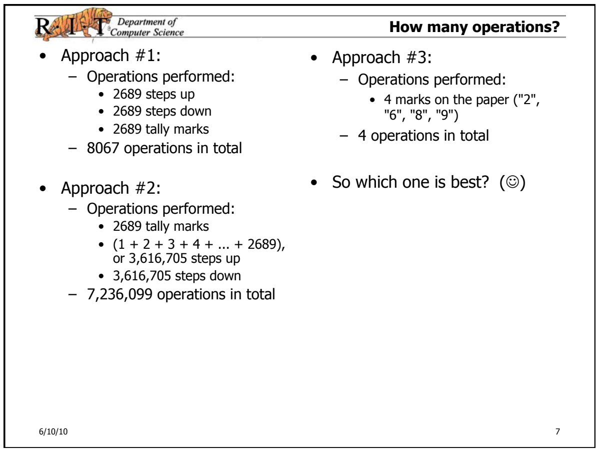
Approach #3

- You ask someone who works at the Tower's information center:
  - They tell you that there are 2689 steps.
  - You take the pen and write each number down on the paper, and give it to your friend.



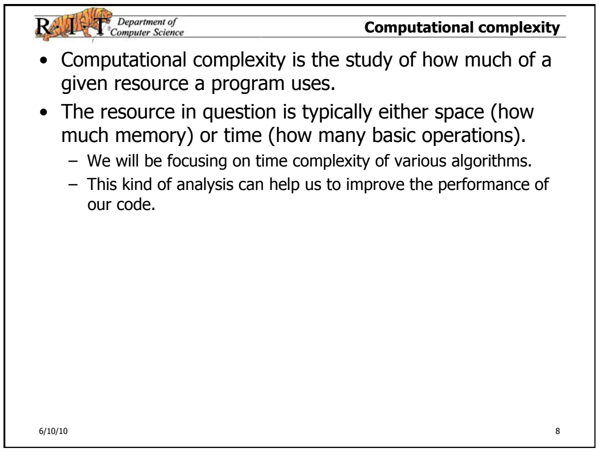
Computing number of operations

- We want to figure out how many operations are required.
  - We *could* just time the procedures, but that can be affected by all kinds of external factors (crowding, etc.). So, we'll make a few assumptions.
- Core assumptions:
  - Walking up or down a step is a single operation.
  - Making a mark on a piece of paper (used to keep a tally, or write numbers) is a single operation.



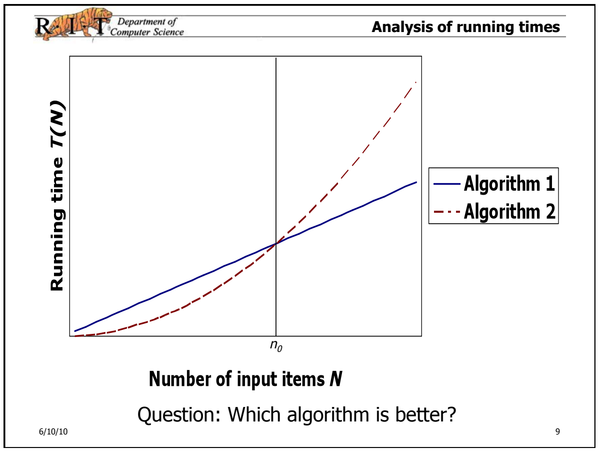
How many operations?

- Approach #1:
  - Operations performed:
    - 2689 steps up
    - 2689 steps down
    - 2689 tally marks
  - 8067 operations in total
- Approach #2:
  - Operations performed:
    - 2689 tally marks
    - (1 + 2 + 3 + 4 + ... + 2689), or 3,616,705 steps up
    - 3,616,705 steps down
  - 7,236,099 operations in total
- Approach #3:
  - Operations performed:
    - 4 marks on the paper ("2", "6", "8", "9")
  - 4 operations in total
- So which one is best? (☺)



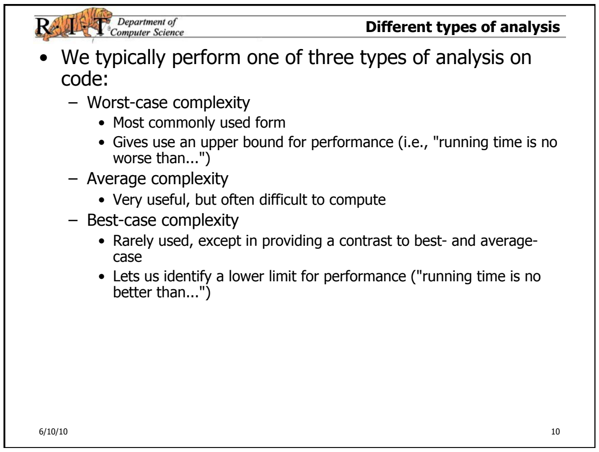
Computational complexity

- Computational complexity is the study of how much of a given resource a program uses.
- The resource in question is typically either space (how much memory) or time (how many basic operations).
  - We will be focusing on time complexity of various algorithms.
  - This kind of analysis can help us to improve the performance of our code.



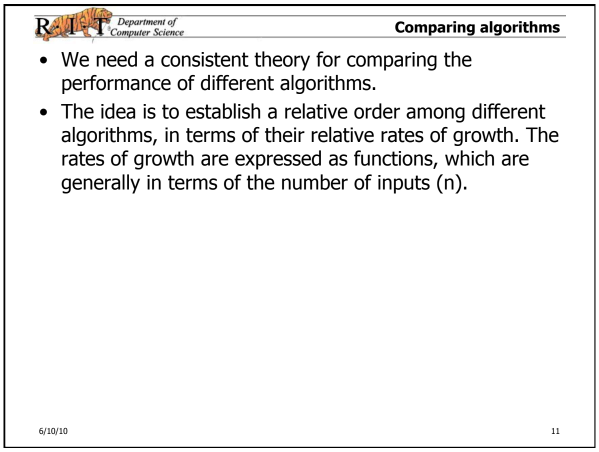
Analysis of running times

Question: Which algorithm is better?



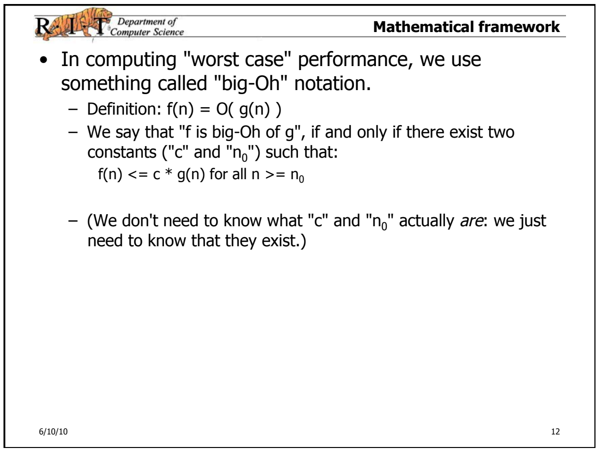
Different types of analysis

- We typically perform one of three types of analysis on code:
  - Worst-case complexity
    - Most commonly used form
    - Gives use an upper bound for performance (i.e., "running time is no worse than...")
  - Average complexity
    - Very useful, but often difficult to compute
  - Best-case complexity
    - Rarely used, except in providing a contrast to best- and average-case
    - Lets us identify a lower limit for performance ("running time is no better than...")



Comparing algorithms

- We need a consistent theory for comparing the performance of different algorithms.
- The idea is to establish a relative order among different algorithms, in terms of their relative rates of growth. The rates of growth are expressed as functions, which are generally in terms of the number of inputs (n).

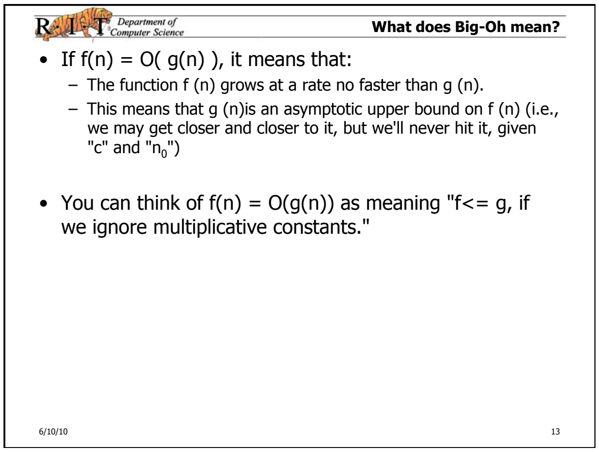


Mathematical framework

- In computing "worst case" performance, we use something called "big-Oh" notation.
  - Definition: f(n) = O(g(n))
  - We say that "f is big-Oh of g", if and only if there exist two constants ("c" and "n_0") such that:

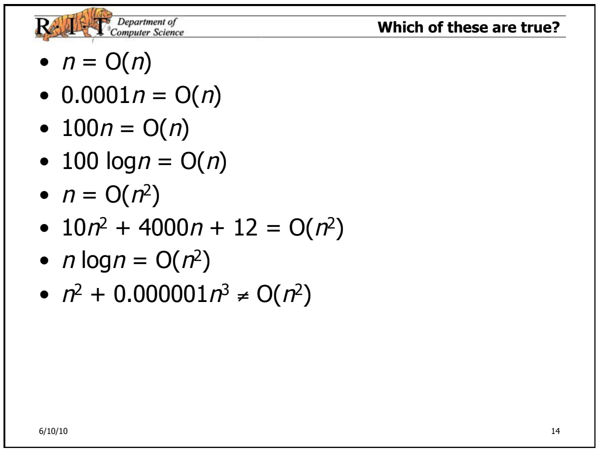
    $f(n) <= c * g(n)$ for all $n >= n_0$

  - (We don't need to know what "c" and "n_0" actually *are*: we just need to know that they exist.)


Department of Computer Science

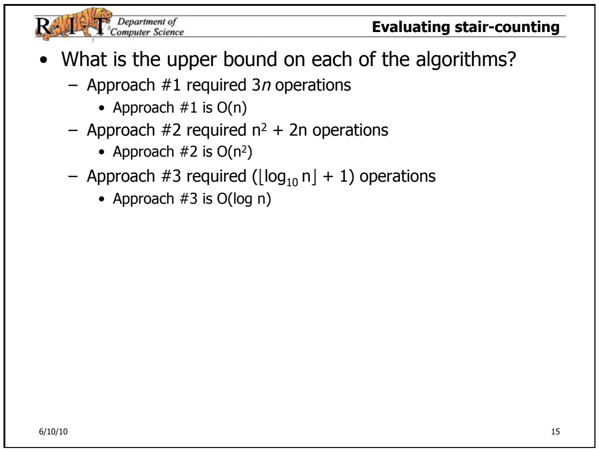
What does Big-Oh mean?

- If f(n) = O(g(n)), it means that:
  - The function f (n) grows at a rate no faster than g (n).
  - This means that g (n)is an asymptotic upper bound on f (n) (i.e., we may get closer and closer to it, but we'll never hit it, given "c" and "n_0")
- You can think of f(n) = O(g(n)) as meaning "f<= g, if we ignore multiplicative constants."


Department of Computer Science

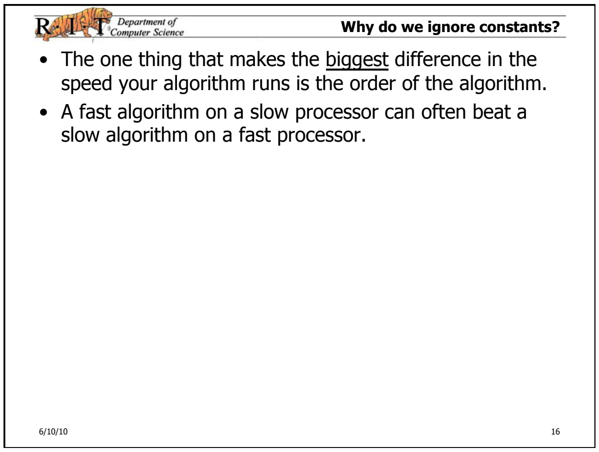
Which of these are true?

- $n = O(n)$
- $0.0001n = O(n)$
- $100n = O(n)$
- $100 \log n = O(n)$
- $n = O(n^2)$
- $10n^2 + 4000n + 12 = O(n^2)$
- $n \log n = O(n^2)$
- $n^2 + 0.000001n^3 \neq O(n^2)$


Department of Computer Science

Evaluating stair-counting

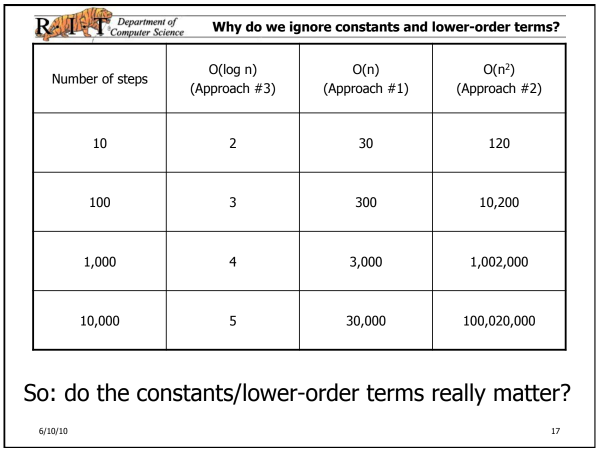
- What is the upper bound on each of the algorithms?
  - Approach #1 required $3n$ operations
    - Approach #1 is O(n)
  - Approach #2 required $n^2 + 2n$ operations
    - Approach #2 is $O(n^2)$
  - Approach #3 required $(\lfloor \log_{10} n \rfloor + 1)$ operations
    - Approach #3 is O(log n)



Department of Computer Science

Why do we ignore constants?

- The one thing that makes the biggest difference in the speed your algorithm runs is the order of the algorithm.
- A fast algorithm on a slow processor can often beat a slow algorithm on a fast processor.

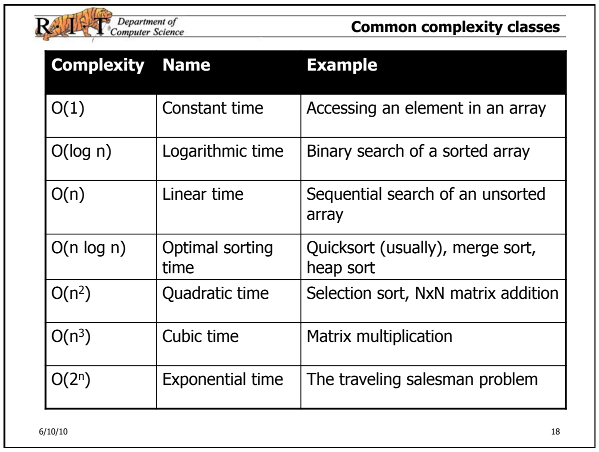


Department of Computer Science

Why do we ignore constants and lower-order terms?

Number of steps	O(log n) (Approach #3)	O(n) (Approach #1)	$O(n^2)$ (Approach #2)
10	2	30	120
100	3	300	10,200
1,000	4	3,000	1,002,000
10,000	5	30,000	100,020,000

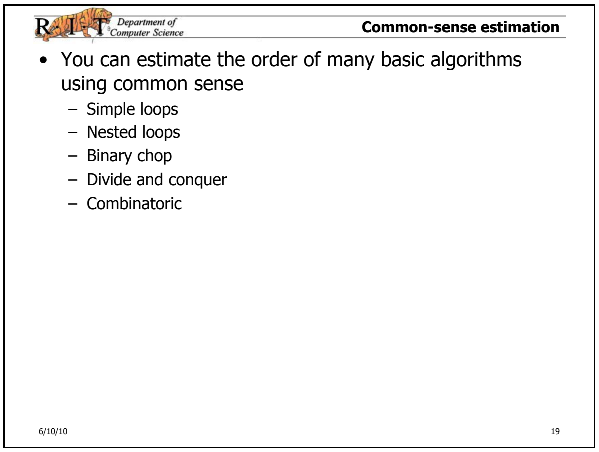
So: do the constants/lower-order terms really matter?



Department of Computer Science

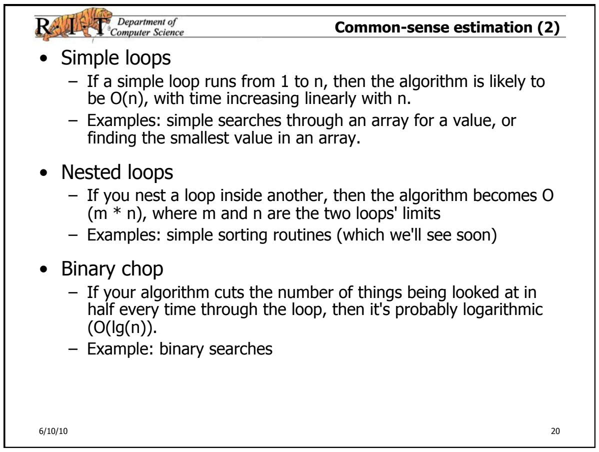
Common complexity classes

Complexity	Name	Example
O(1)	Constant time	Accessing an element in an array
O(log n)	Logarithmic time	Binary search of a sorted array
O(n)	Linear time	Sequential search of an unsorted array
O(n log n)	Optimal sorting time	Quicksort (usually), merge sort, heap sort
$O(n^2)$	Quadratic time	Selection sort, NxN matrix addition
$O(n^3)$	Cubic time	Matrix multiplication
$O(2^n)$	Exponential time	The traveling salesman problem



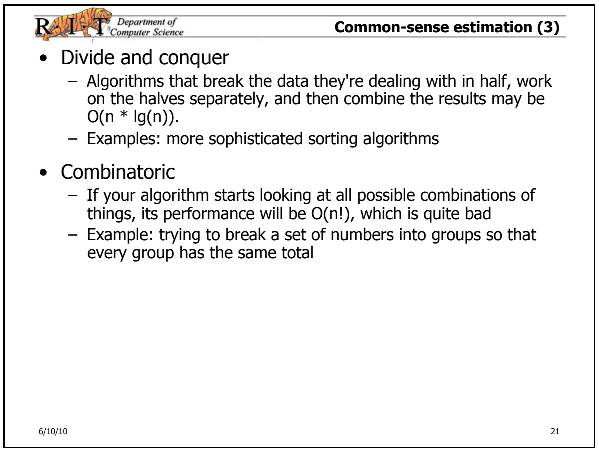
Common-sense estimation

- You can estimate the order of many basic algorithms using common sense
  - Simple loops
  - Nested loops
  - Binary chop
  - Divide and conquer
  - Combinatoric



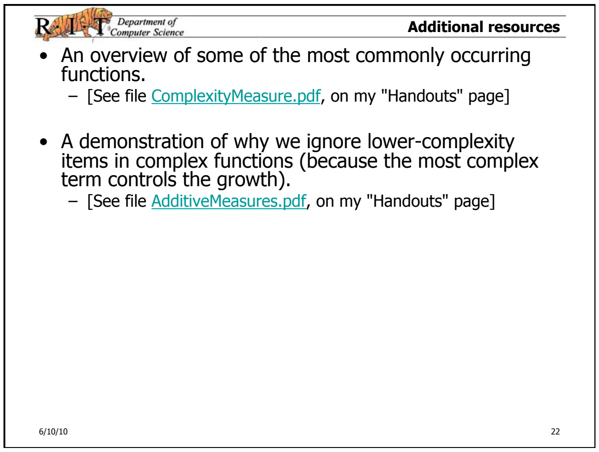
Common-sense estimation (2)

- Simple loops
  - If a simple loop runs from 1 to n, then the algorithm is likely to be O(n), with time increasing linearly with n.
  - Examples: simple searches through an array for a value, or finding the smallest value in an array.
- Nested loops
  - If you nest a loop inside another, then the algorithm becomes O (m * n), where m and n are the two loops' limits
  - Examples: simple sorting routines (which we'll see soon)
- Binary chop
  - If your algorithm cuts the number of things being looked at in half every time through the loop, then it's probably logarithmic (O(lg(n)).
  - Example: binary searches



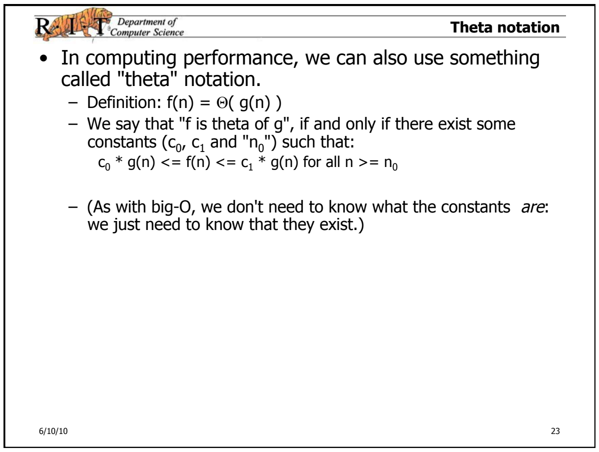
Common-sense estimation (3)

- Divide and conquer
  - Algorithms that break the data they're dealing with in half, work on the halves separately, and then combine the results may be O(n * lg(n)).
  - Examples: more sophisticated sorting algorithms
- Combinatoric
  - If your algorithm starts looking at all possible combinations of things, its performance will be O(n!), which is quite bad
  - Example: trying to break a set of numbers into groups so that every group has the same total



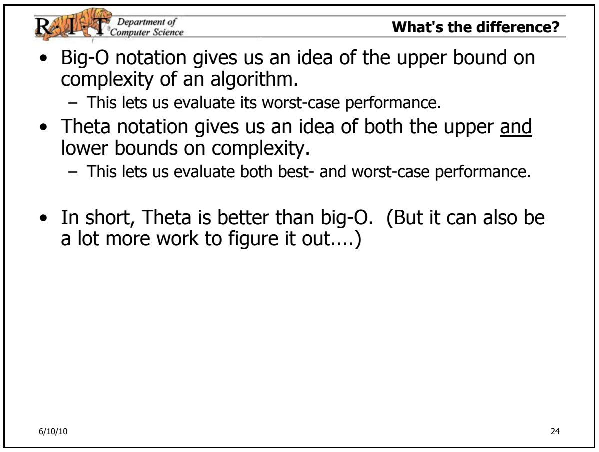
Additional resources

- An overview of some of the most commonly occurring functions.
  - [See file ComplexityMeasure.pdf, on my "Handouts" page]
- A demonstration of why we ignore lower-complexity items in complex functions (because the most complex term controls the growth).
  - [See file AdditiveMeasures.pdf, on my "Handouts" page]



Theta notation

- In computing performance, we can also use something called "theta" notation.
  - Definition: f(n) = Θ(g(n))
  - We say that "f is theta of g", if and only if there exist some constants (c_0, c_1 and "n_0") such that:
    $c_0 * g(n) <= f(n) <= c_1 * g(n)$ for all $n >= n_0$
  - (As with big-O, we don't need to know what the constants *are*: we just need to know that they exist.)



What's the difference?

- Big-O notation gives us an idea of the upper bound on complexity of an algorithm.
  - This lets us evaluate its worst-case performance.
- Theta notation gives us an idea of both the upper and lower bounds on complexity.
  - This lets us evaluate both best- and worst-case performance.
- In short, Theta is better than big-O. (But it can also be a lot more work to figure it out....)