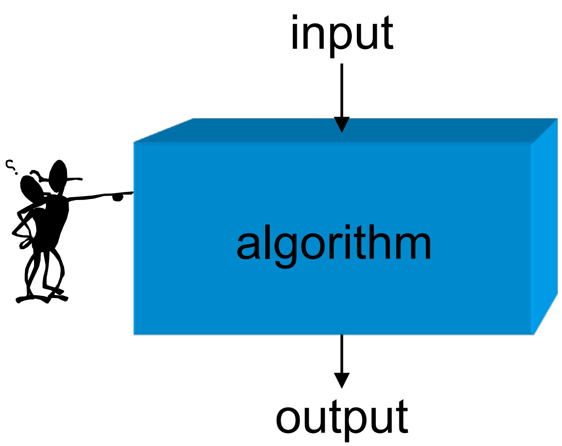
Lecture 3:
Algorithms and Complexity

Bioalgorithms Fall 2011

Study Chapter 2.1-2.8

What is an algorithm?

- An algorithm is a sequence of instructions that one must perform in order to solve a well-formulated problem.



Problem: Complexity

Algorithm: Correctness
Complexity

Problem: Buying Textbook with Credit Card

Algorithm #1:

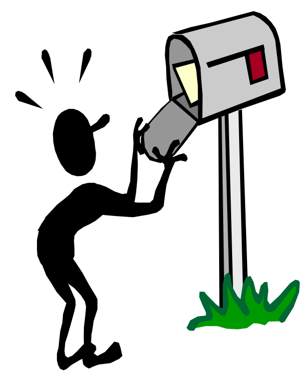
1. Go to the bookstore at the student union.
2. Find the shelf with the tag "COMP 590-87" or "COMP 790-87".
3. Take a copy of the book.
4. Go to the register.
5. Check out using credit card.
6. Walk out with book

Algorithm #2:

1. Go to Amazon.com
2. Search for the book entitled "An Introduction to Bioinformatics Algorithms".
3. Click "Add to shopping cart".
4. Click "Proceed to checkout".
5. Sign in your account.
6. Fill the shipping information.
7. Fill in the credit card and billing information.
8. Place the order.
9. Wait 5-10 days for book to arrive

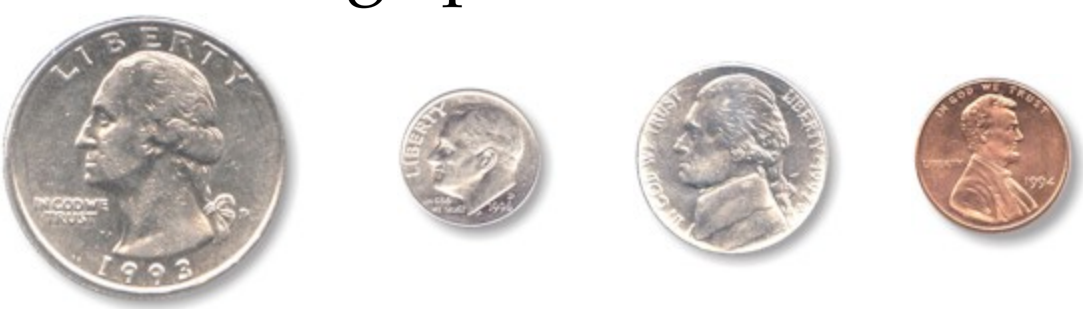
Two observations

- Given a problem, there may be many correct algorithms.
  - They give identical outputs for the same inputs
  - They give the expected outputs for any valid input
- The costs to perform different algorithms may be different.
  - Some are faster (i.e. get the book immediately, or you wait for a week)
  - Some are less expensive



Correctness

- An algorithm is correct only if it produces correct result for all input instances.
  - If the algorithm gives an incorrect answer for one or more input instances, it is an incorrect algorithm.
- Coin change problem
  - Input: an amount of money M in cents
  - Output: the smallest number of coins
- US coin change problem



US Coin Change

72 cents

↓

US Coin Change

↓

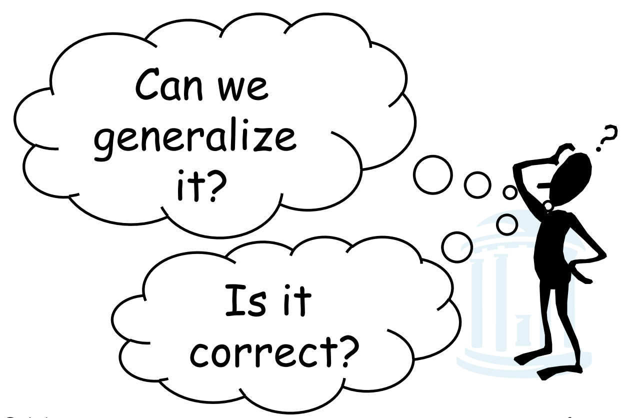
Two quarters, 22 cents left

Two dimes, 2 cents left

Two pennies

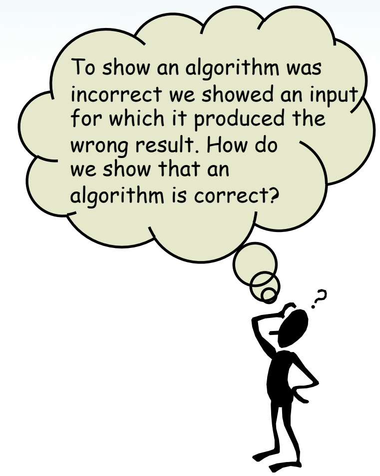
Classic Algorithm

$$r \leftarrow M$$
$$q \leftarrow r/25$$
$$r \leftarrow r - 25 \cdot q$$
$$d \leftarrow r/10$$
$$r \leftarrow r - 10 \cdot d$$
$$n \leftarrow r/5$$
$$r \leftarrow r - 5 \cdot n$$
$$p \leftarrow r$$



Change Problem

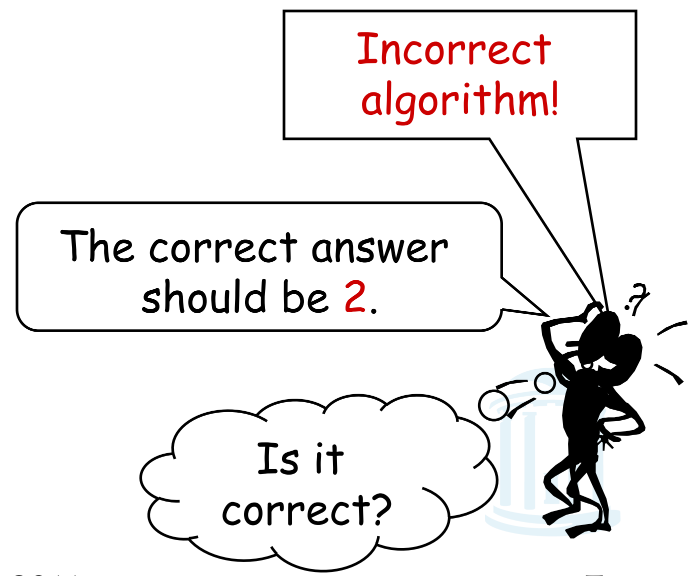
- Input:
  - an amount of money M
  - an array of denominations $c = (c_1, c_2, \ldots, c_d)$ in order of decreasing value
- Output: the smallest number of coins



$M = 40$
$c = (25, 20, 10, 5, 1)$ →

$$
\begin{array}{l}
r \leftarrow M \\
n \leftarrow 0 \\
\text{for } k \leftarrow 1 \text{ to } d \\
\quad i_k \leftarrow r / c_k \\
\quad n \leftarrow n + i_k \\
\quad r \leftarrow r - c_k \times i_k \\
\text{return } n
\end{array}
$$

→ 3



How to Compare Algorithms?

- Complexity – the cost of an algorithm can be measured in either time and space
  - Correct algorithms may have different complexities.
- How do we assign "cost" for time?
- The cost to perform an instruction may vary dramatically.
  - An instruction may be an algorithm itself.
  - The complexity of an algorithm is NOT equivalent to the number of instructions.
- How to analyze an algorithm's complexity
  - An aside: Algorithm "Styles"

Ex Style: Recursive Algorithms

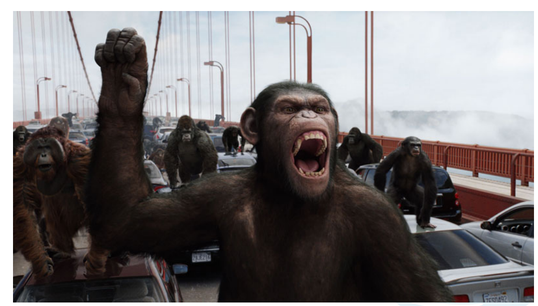
- Recursion is a technique for describing functions in terms of themselves.
  - These recursive calls are to simpler versions of the original function.
  - The simplest versions, called base cases, are merely declared.

    Recursive definition: $\text{factorial}(n) = n \times \text{factorial}(n-1)$

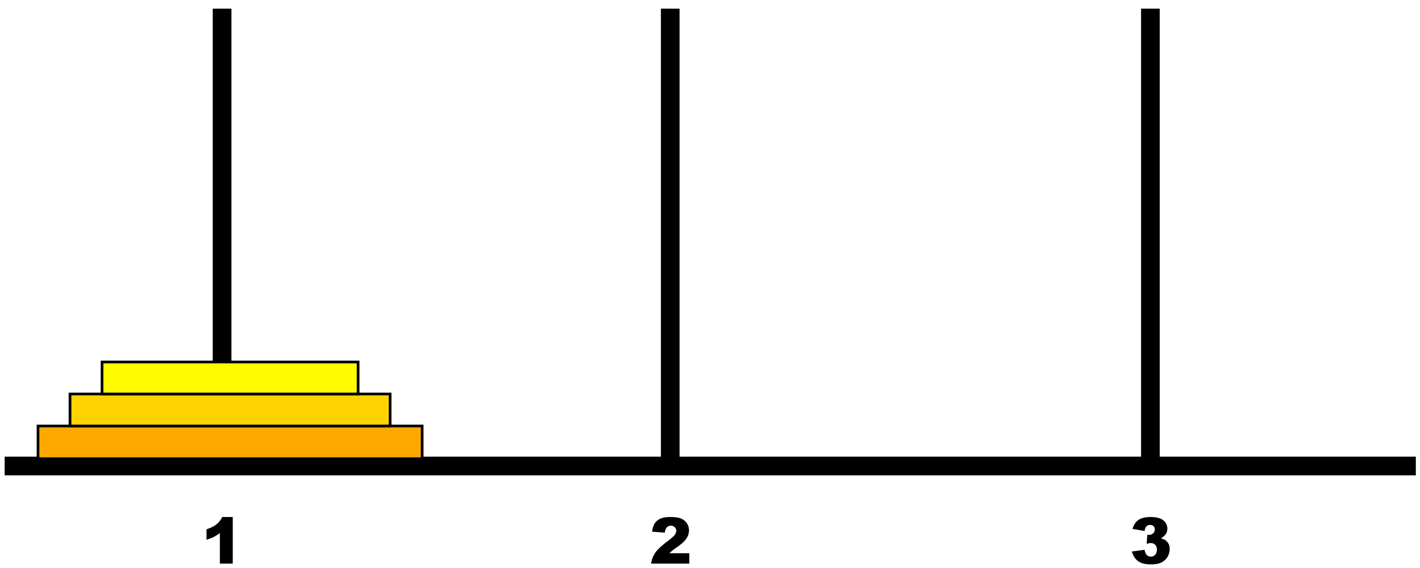
    Base case: $\text{factorial}(1) = 1$
  - Easy to analyze
- Thinking recursively…

Towers of Hanoi

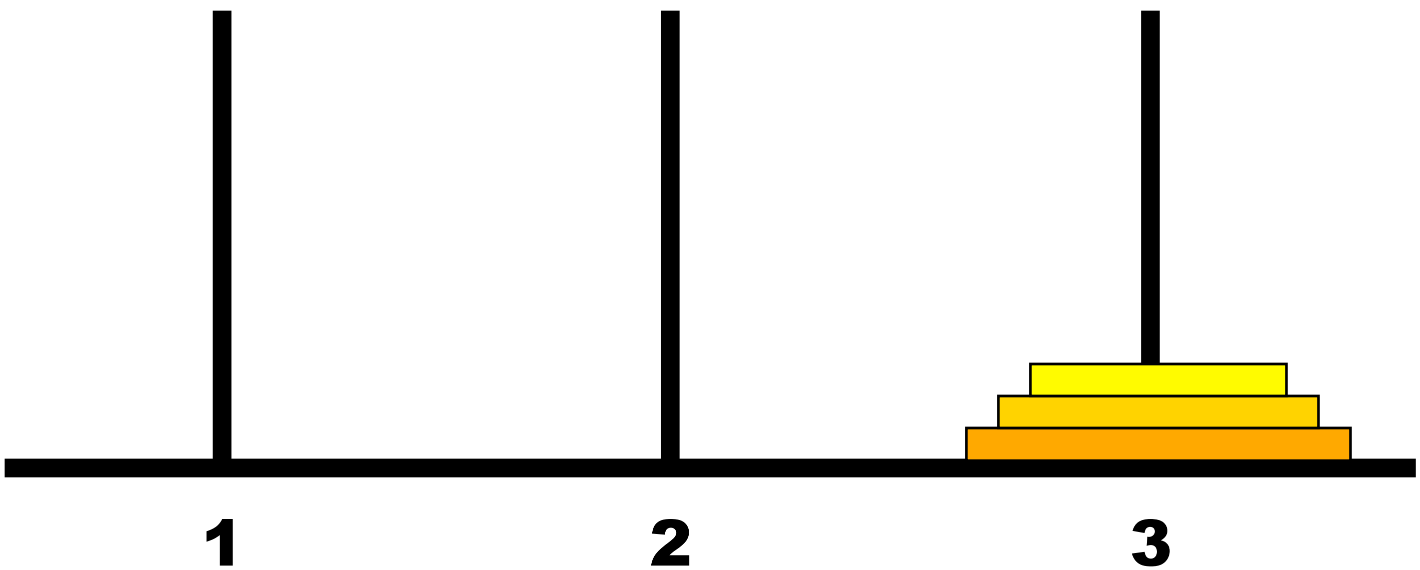
- There are three pegs and a number of disks with decreasing radii (smaller ones on top of larger ones) stacked on Peg 1.
- Goal: move all disks to Peg 3.
- Rules:
  - When a disk is moved from one peg it must be placed on another peg.
  - Only one disk may be moved at a time, and it must be the top disk on a tower.
  - A larger disk may never be placed upon a smaller disk.



A three disk tower



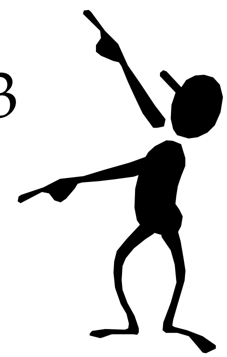
Simplifying the algorithm for 3 disks



The problem for N disks becomes

- A base case of a one-disk move.
- A recursive step for moving n-1 disks.

- To move n disks from Peg 1 to Peg 3, we need to
  - Move (n-1) disks from Peg 1 to Peg 2
  - Move the n^{th} disk from Peg 1 to Peg 3
  - Move (n-1) disks from Peg 2 to Peg 3
  - The number of disk moves is



We move the n-1 stack twice

$$T(1) = 1$$

$$T(n) = 2T(n-1) + 1 = 2^n - 1$$

Exponential algorithm

Towers of Hanoi

- If you play HanoiTowers with . . . it takes . . .
  - 1 disk … 1 move
  - 2 disks … 3 moves
  - 3 disks … 7 moves
  - 4 disks … 15 moves
  - 5 disks … 31 moves
  - .
  - .
  - .
  - 20 disks . . . 1,048,575 moves
  - 32 disks . . . 4,294,967,295 moves

Sorting

- A very common problem is to arrange data into either ascending or descending order
  - **Viewing, printing**
  - **Faster to search, find min/max, compute median/mode, etc.**
- Lots of sorting algorithms
  - **From the simple to very complex**
  - **Some optimized for certain situations (lots of duplicates, almost sorted, etc.)**

Selection Sort

Find the smallest element and swap it with the first:	27	12	3	18	11	7
Find the next smallest element and swap it with the second:	3	12	27	18	11	7
Do the same for the third element:	3	7	27	18	11	12
And the fourth:	3	7	11	18	27	12
Finally, the fifth:	3	7	11	12	27	18
Completely sorted:	3	7	11	12	18	27

"In-place" sort

Selection sort

```
def selectionSortRecursive(a,first,last):
    if (first < last):
        index = indexOfMin(a,first,last)
        temp = a[index]
        a[index] = a[first]
        a[first] = temp
        a = selectionSortRecursive(a,first+1,last)
    return a
```

$(n - 1)$ swaps

Quadratic in time

$$\frac{n(n-1)}{2} - 1$$ comparisons

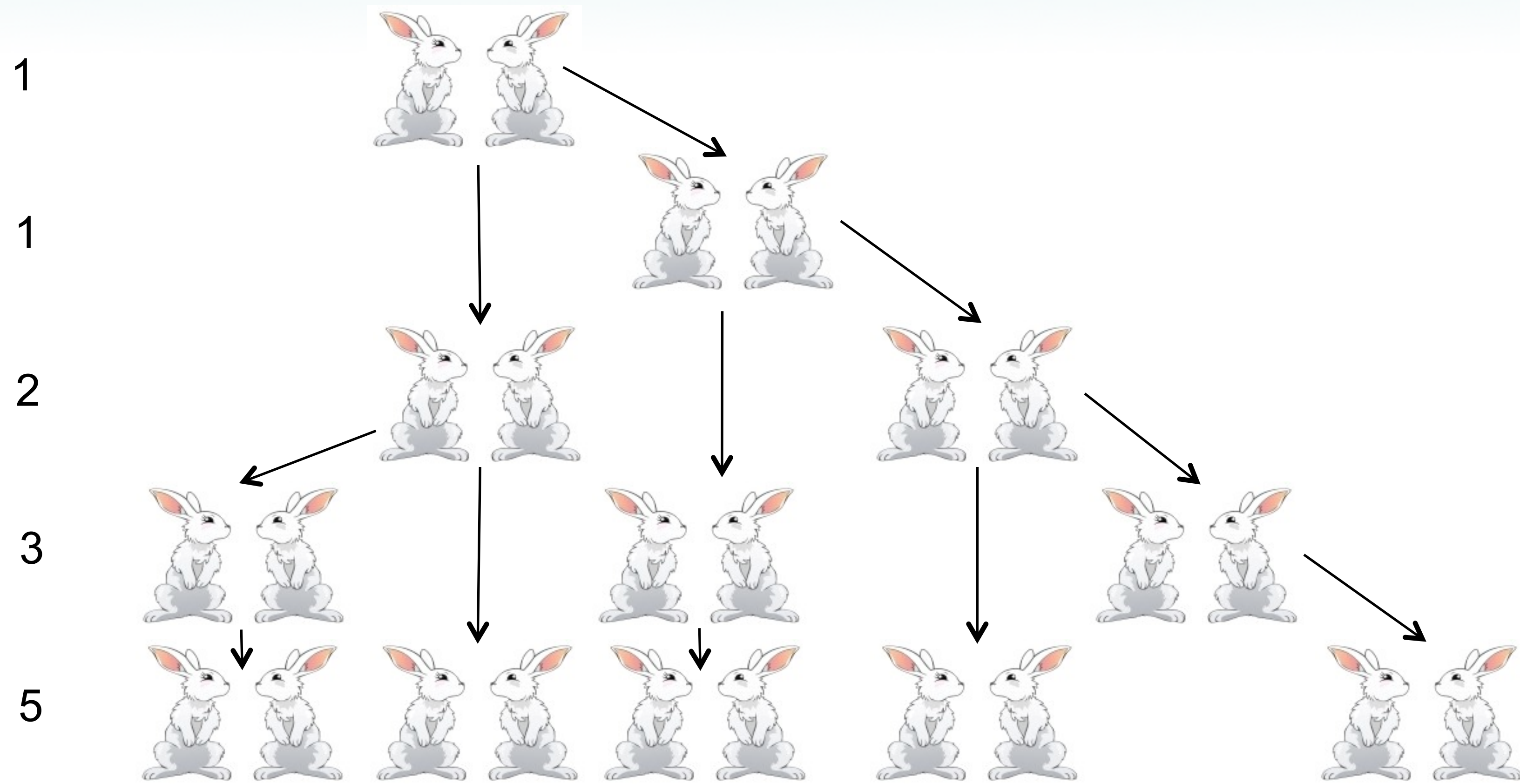
```
def indexOfMin(arr,first,last):
    index = first
    for k in xrange(index+1,last):
        if (arr[k] < arr[index]):
            index = k
    return index
```

Year 1202: Leonardo Fibonacci:

- He asked the following question:
  - How many pairs of rabbits are produced from a single pair in one year if every year each pair of rabbits more than 1 year old produces a new pair?

  - Here we assume that each pair has one male and one female, and each pair lives long enough to have two litters, and initially we have one pair
  - $f(n)$: the number of “breeding” pairs present at the beginning of year n

Fibonacci Number



Fibonacci Number

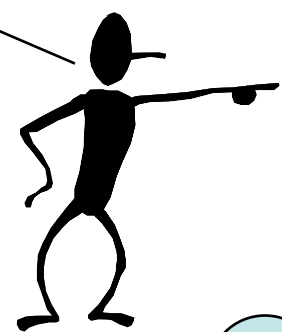
- Clearly, we have:
  - $f(1) = 1$ (the first pair we have)
  - $f(2) = 1$ (still only the first pair we have because they are just 1 month old. They need to be more than one month old to reproduce)
  - $f(n) = f(n\text{-}1) + f(n\text{-}2)$ because $f(n)$ is the sum of the old rabbits from last month ($f(n\text{-}1)$) and the new rabbits reproduced from those $f(n\text{-}2)$ rabbits who are now old enough to reproduce.
  - f: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
  - The solution for this recurrence is:

$$f(n) = \tfrac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

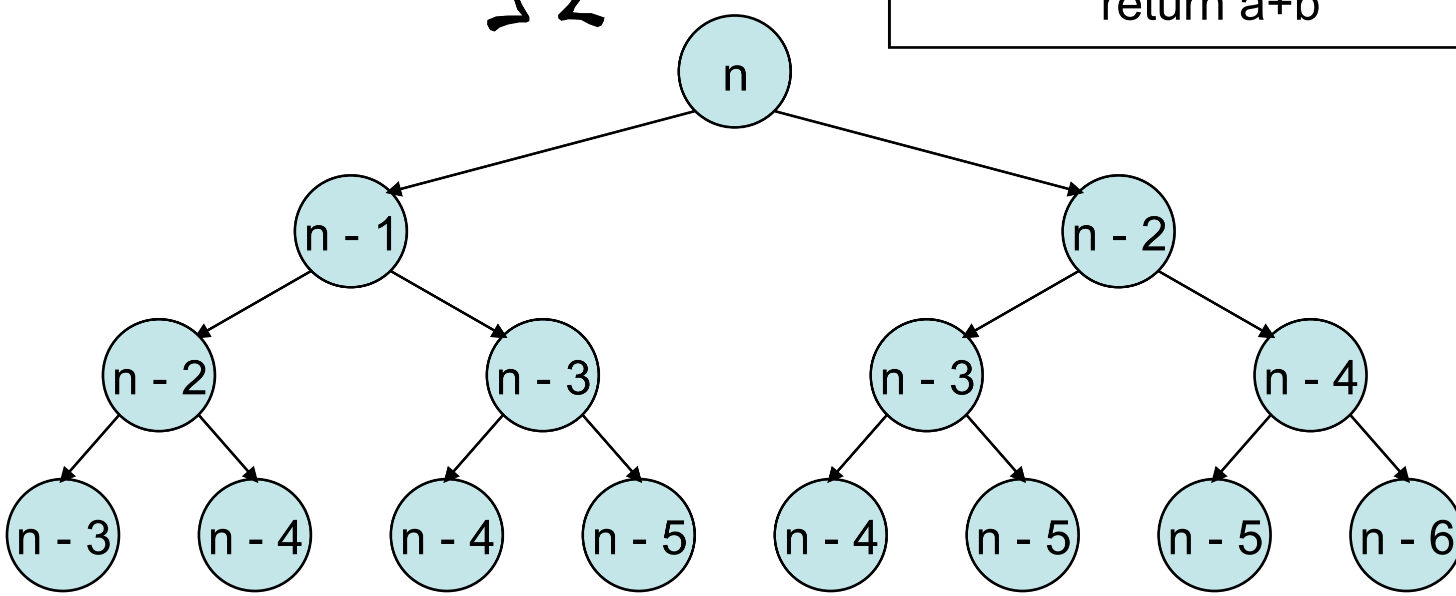
Fibonacci Number

Exponential time!

Recursive Algorithm



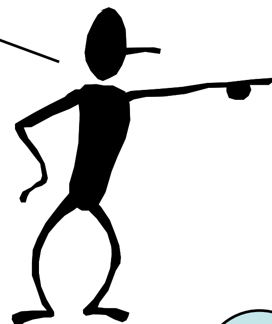
```
def fibonacciRecursive(n):
        if (n <= 2):
                return 1
       else:
                a = fibonacciRecursive(n-1)
                b = fibonacciRecursive(n-2)
                return a+b
```



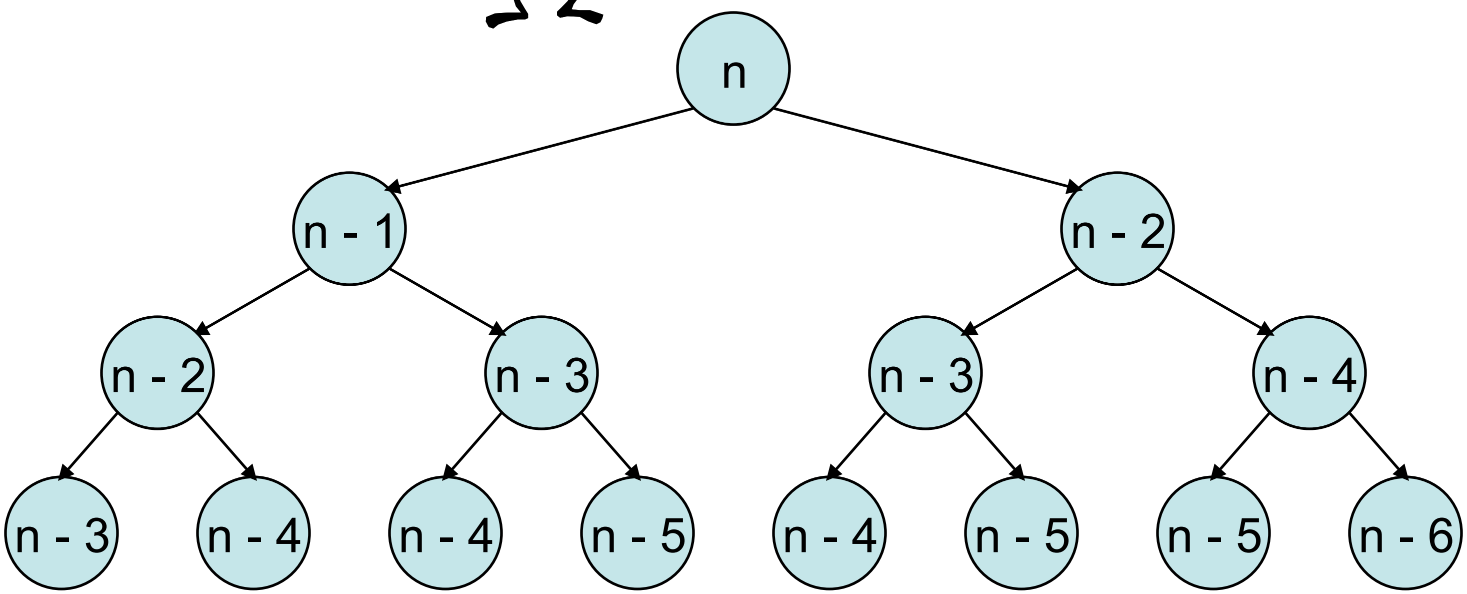
Fibonacci Number

Linear time!

Iterative Algorithm



```
def fibonacci(n):
    f0, f1 = 0, 1
    for i in xrange(n):
        f0, f1 =f1, f0 + f1
    return f0
```



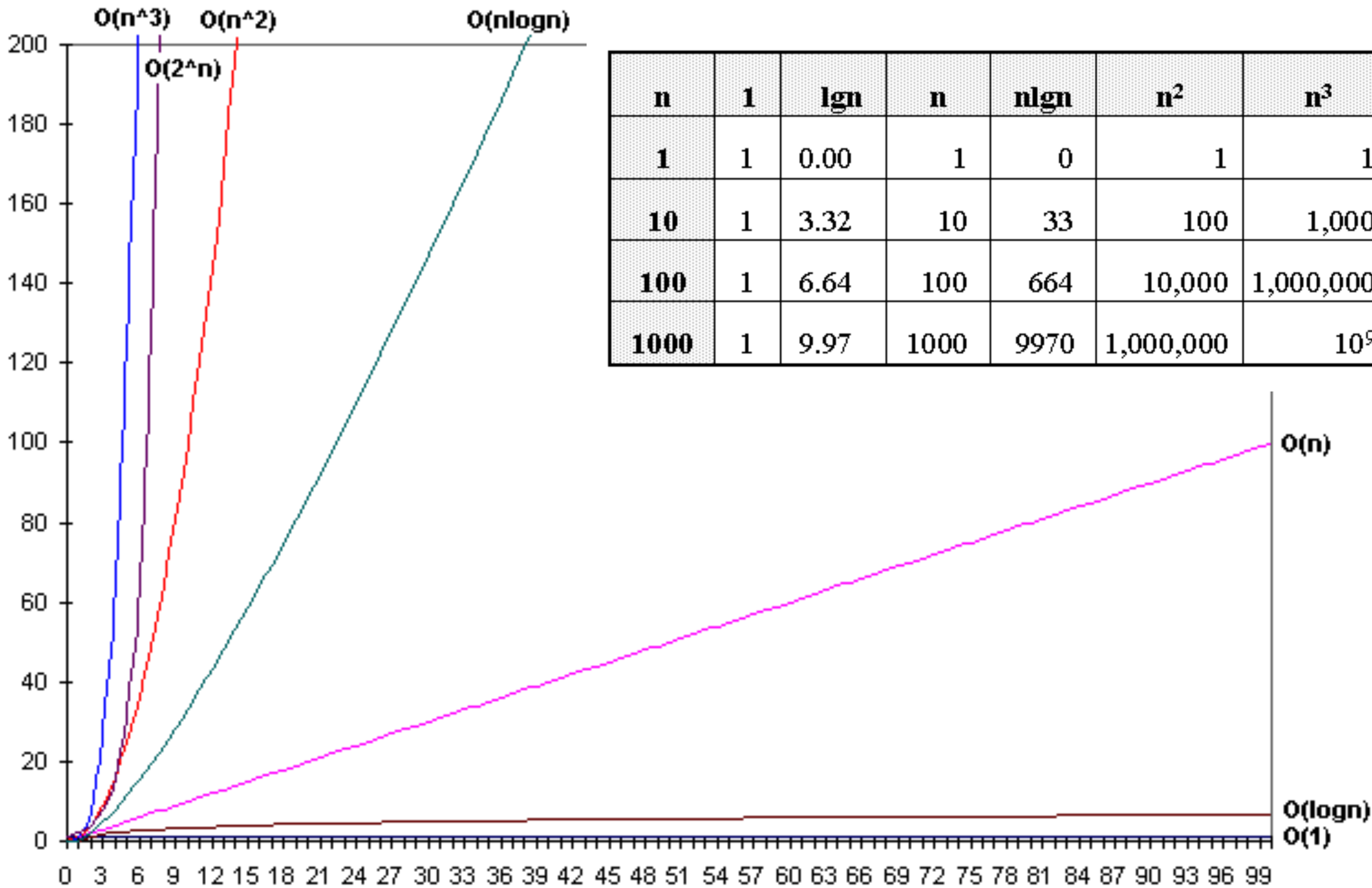
Is there a “real difference”?

- 10^1
- 10^2 Number of students in computer science department
- 10^3 Number of students in the college of art and science
- 10^4 Number of students enrolled at UNC
- …
- …
- 10^10 Number of stars in the galaxy
- 10^20 Total number of all stars in the universe
- 10^80 Total number of particles in the universe
- 10^100 << Number of moves needed for 400 disks in the Towers of Hanoi puzzle

- Towers of Hanoi puzzle is *computable* but it is NOT feasible.

Is there a "real" difference?

- Growth of functions



n	1	lgn	n	nlgn	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1,000	1024
100	1	6.64	100	664	10,000	1,000,000	1.2×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.1×10^{301}

Asymptotic Notation

- *Order of growth* is the interesting measure:
  - Highest-order term is what counts
    - As the input size grows larger it is the high order term that dominates
- Θ notation: $\Theta(n^2)$ = "this function grows similarly to n^2".
- Big-O notation: O (n^2) = "this function grows at least as *slowly* as n^2".
  - Describes an *upper bound.*

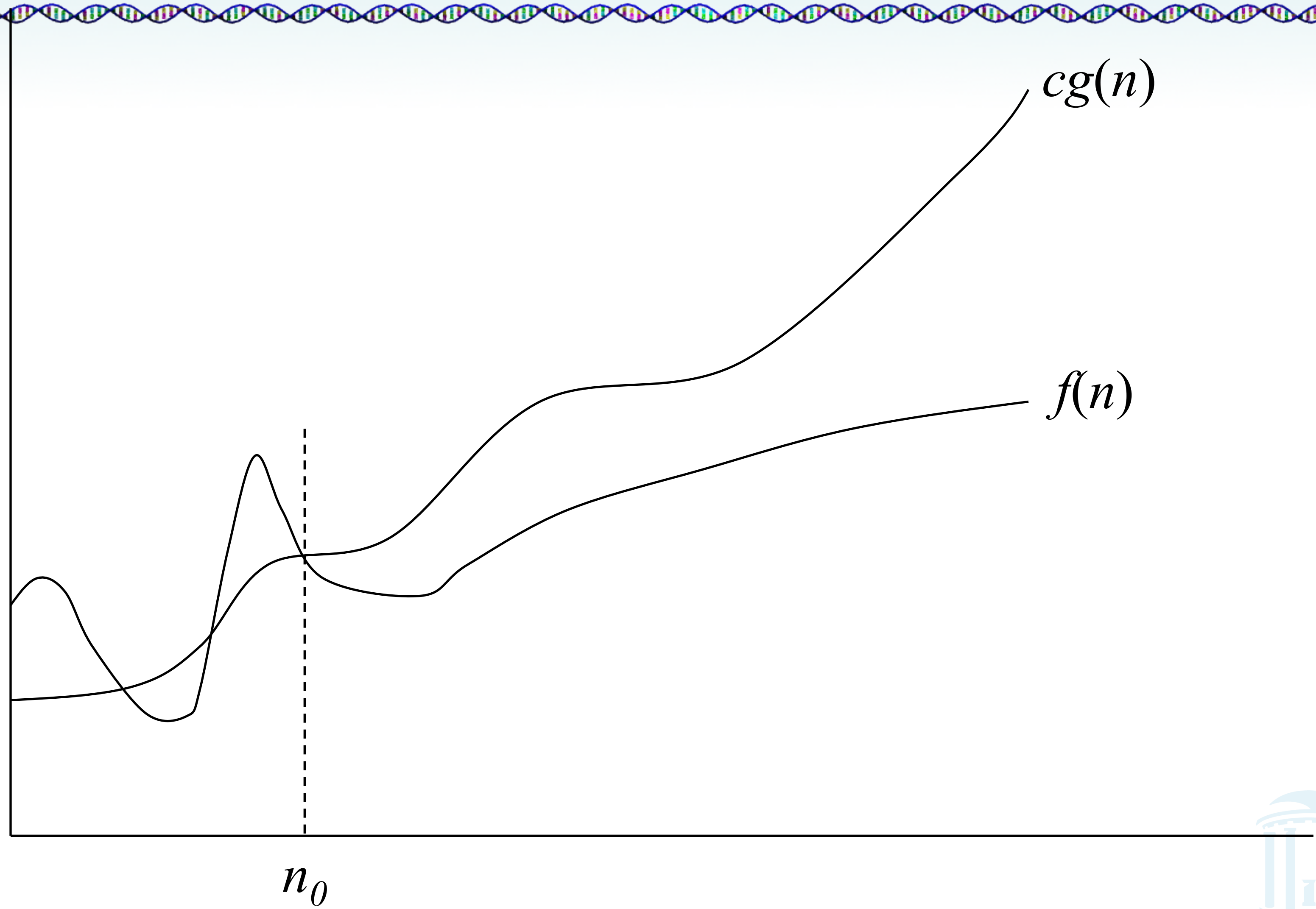
Big-O Notation

$f(n) = O(g(n))$: there exist positive constants c and n_0 such that

$$0 \le f(n) \le cg(n) \text{ for all } n \ge n_0$$

- What does it mean?
  - If $f(n) = O(n^2)$, then:
    - $f(n)$ can be larger than n^2 sometimes, **but…**
    - We can choose some constant c and some value n_0 such that for **every** value of n larger than n_0 : $f(n) < cn^2$
    - That is, for values larger than n_0, $f(n)$ is never more than a constant multiplier greater than n^2
    - Or, in other words, $f(n)$ does not grow more than a constant factor faster than n^2.

Visualization of $O(g(n))$



Big-O Notation

$$2n^2 = O(n^2)$$

$$1{,}000{,}000n^2 + 150{,}000 = O(n^2)$$

$$5n^2 - 7n + 20 = O(n^2)$$

$$2n^3 + 2 \neq O(n^2)$$

$$n^{2.1} \neq O(n^2)$$

Big-O Notation

- Prove that: $20n^2 + 2n + 5 = O(n^2)$
- Let $c = 21$ and $n_0 = 4$
- $21n^2 > 20n^2 + 2n + 5$ for all $n > 4$

  $n^2 > 2n + 5$ for all $n > 4$

  TRUE

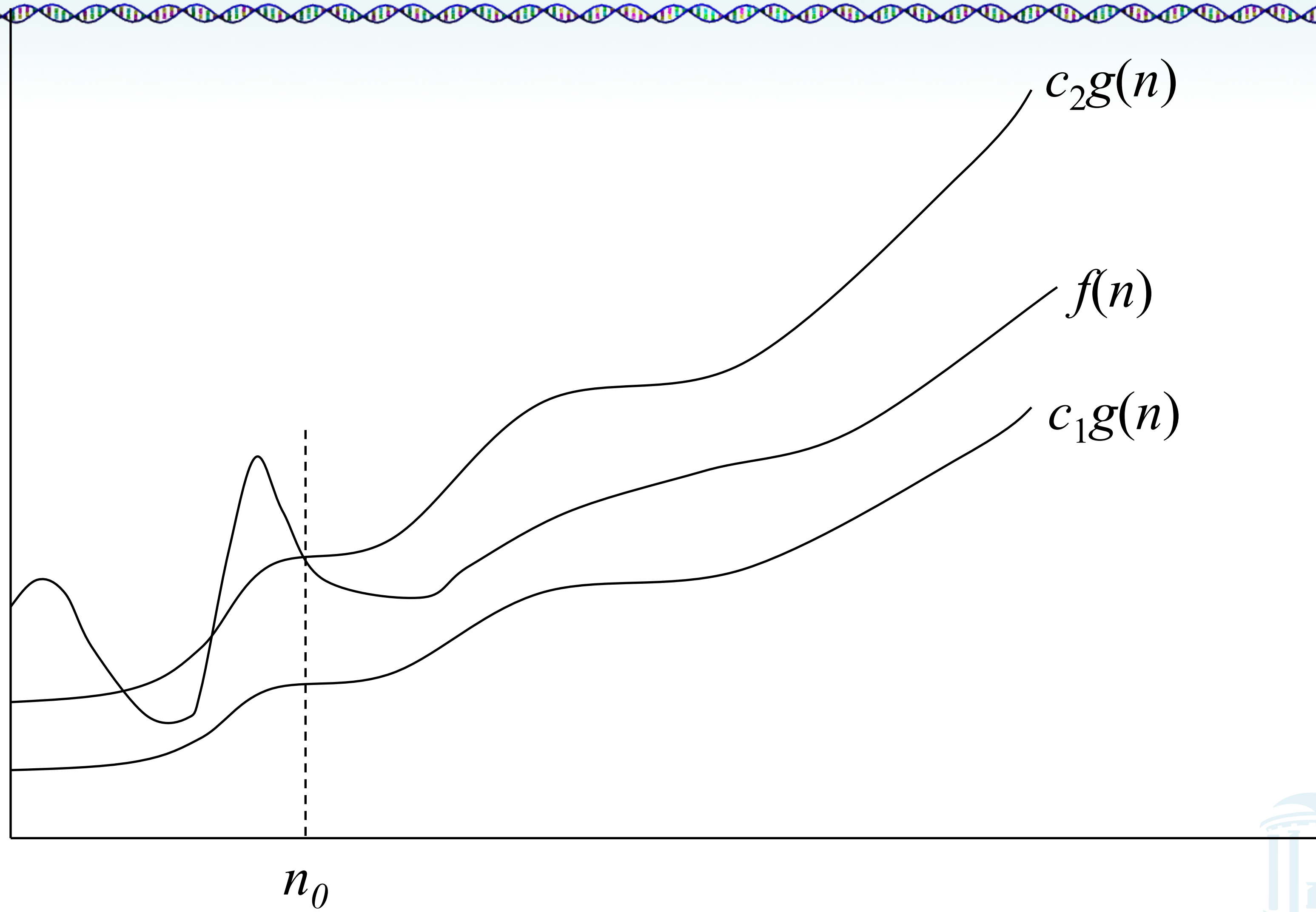
Θ-Notation

- Big-O is not a tight upper bound. In other words $n = O(n^2)$
- Θ provides a tight bound

$$f(n) = \Theta(g(n)) \text{: there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$$
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0$$

- $n = \mathrm{O}(n^2) \ne \Theta(n^2)$
- $200n^2 = \mathrm{O}(n^2) = \Theta(n^2)$
- $n^{2.5} \ne \mathrm{O}(n^2) \ne \Theta(n^2)$

Visualization of $\Theta(g(n))$



Some Other Asymptotic Functions

- Little *o* – A **non-tight** asymptotic upper bound
  - $n = o(n^2)$, $n = O(n^2)$
  - $3n^2 \neq o(n^2)$, $3n^2 = O(n^2)$

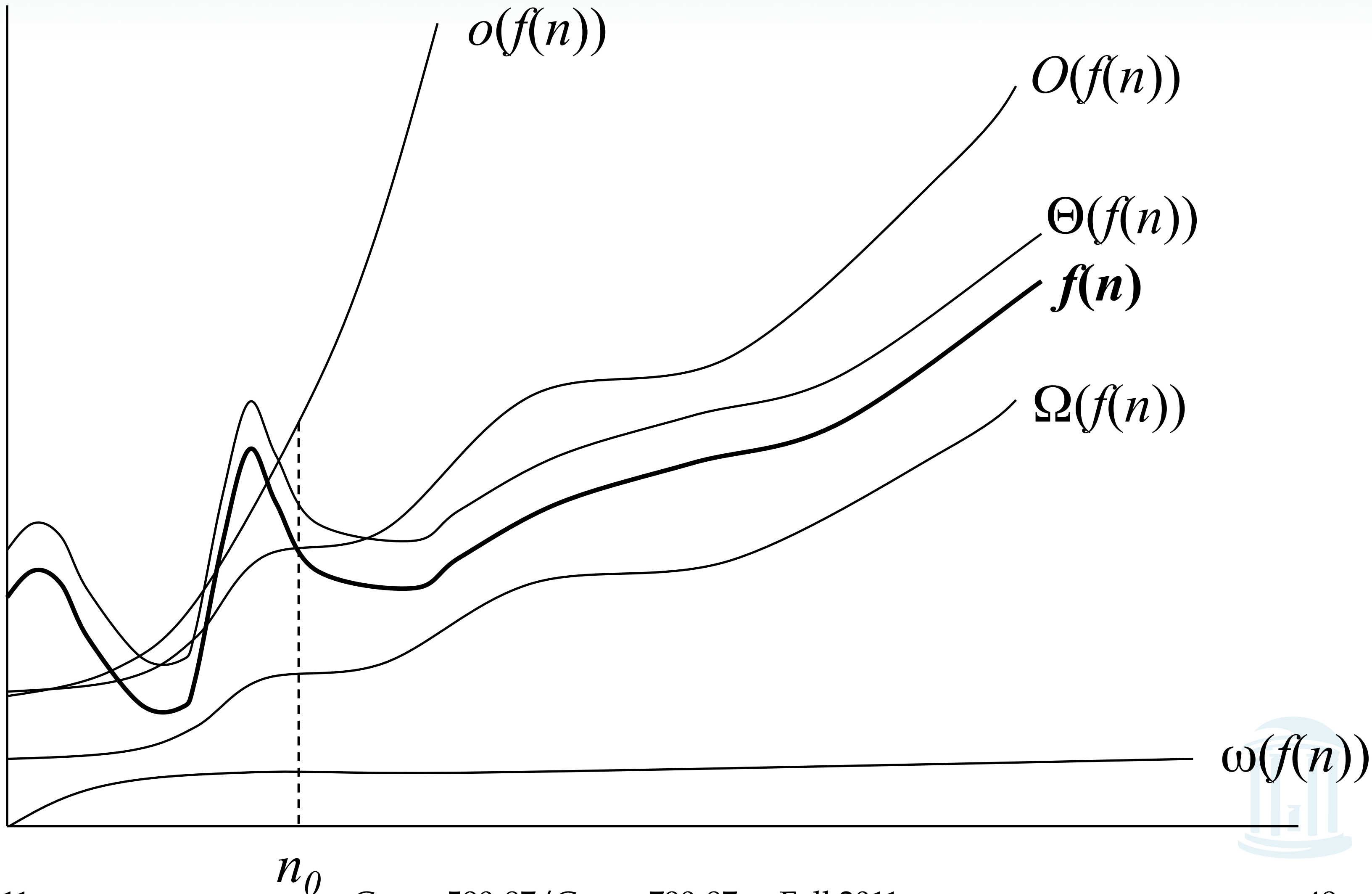
The difference between "big-O" and "little-o" is subtle. For f(n) = O(g(n)) the bound 0 ≤ f(n) ≤ c g(n), n > n_0 holds for *any* c. For f(n) = o(g(n)) the bound 0 ≤ f(n) < c g(n), n > n_0 holds for *all* c.

- Ω – A **lower** bound

$$f(n) = \Omega(g(n))\text{: there exist positive constants } c \text{ and } n_0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq n_0$$

  - $n^2 = \Omega(n)$
- ω – A **non-tight** asymptotic lower bound
- $f(n) = \Theta(n) \Leftrightarrow f(n) = O(n)$ **and** $f(n) = \Omega(n)$

Visualization of Asymptotic Growth



Analogy to Arithmetic Operators

$$f(n) = O(g(n)) \qquad \approx \qquad f \le g$$

$$f(n) = \Omega(g(n)) \qquad \approx \qquad f \ge g$$

$$f(n) = \Theta(g(n)) \qquad \approx \qquad f = g$$

$$f(n) = o(g(n)) \qquad \approx \qquad f < g$$

$$f(n) = \omega(g(n)) \qquad \approx \qquad f > g$$

Measures of complexity

- Best case
  - **Super-fast in some limited situation is not very valuable information**
- Worst case
  - **Good upper-bound on behavior**
  - **Never gets worse than this**
- Average case
  - **Averaged over all possible inputs**
  - **Most useful information about overall performance**
  - **Can be hard to compute precisely**

Complexity

- Time complexity is not necessarily the same as the space complexity
- Space Complexity: how much space an algorithm needs (as a function of *n*)
- Time vs. space

Next Time

- Algorithm "Styles" and design techniques
  - Exhaustive search
  - Greedy algorithms
  - Branch and bound algorithms
  - Dynamic programming
  - Divide and conquer algorithms
  - Randomized algorithms
- Tractable vs intractable algorithms