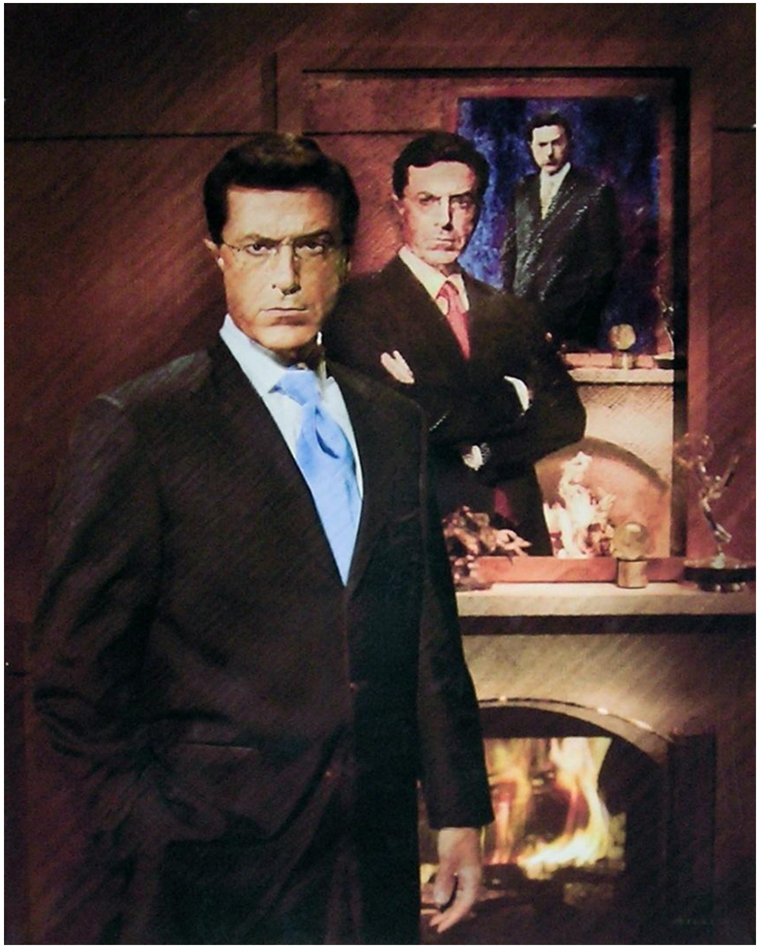
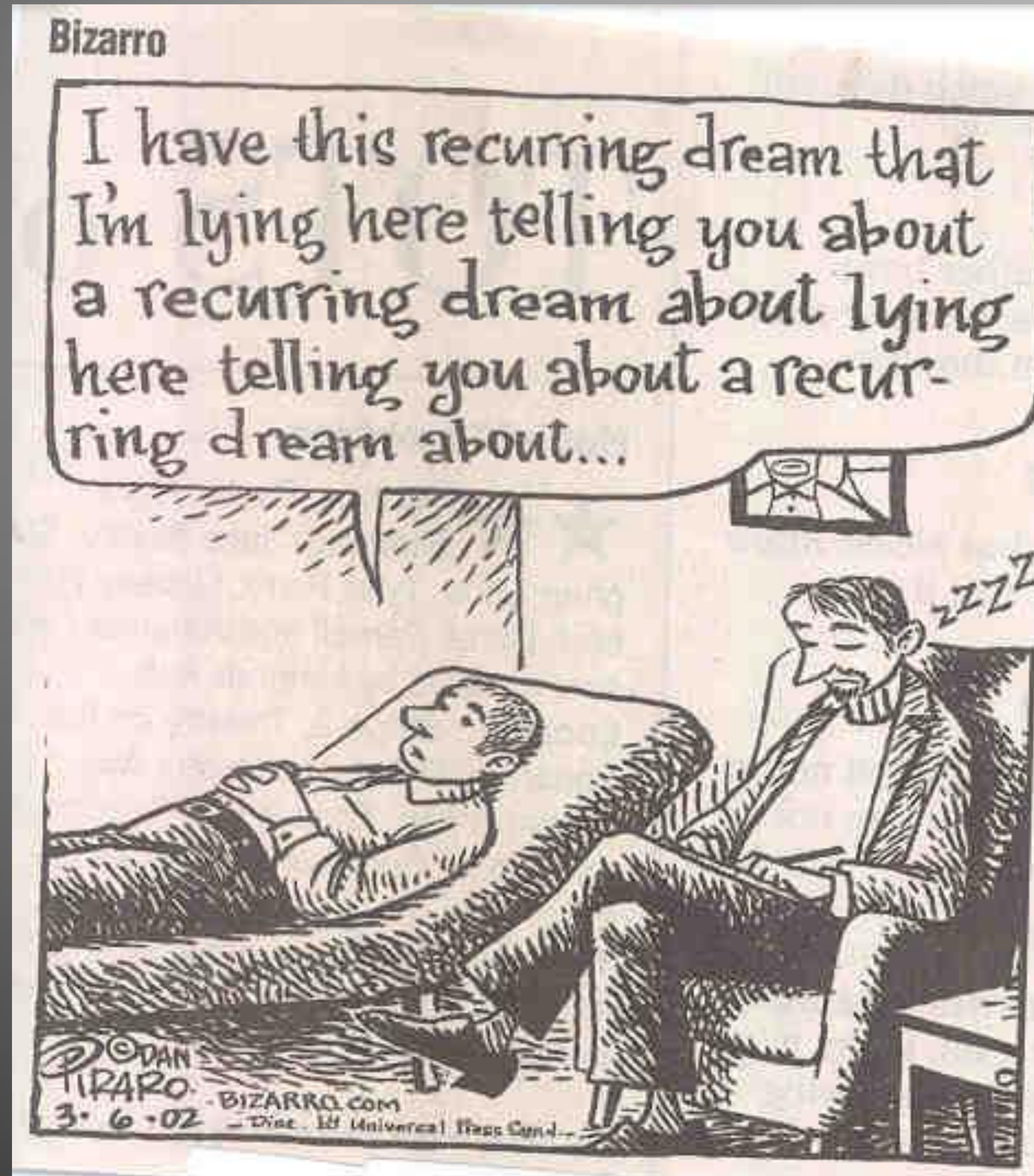


# CSSE 220
# Day 22

Recursion, Efficiency, and the Time-Space Trade Off; Selection Sort and Big-Oh

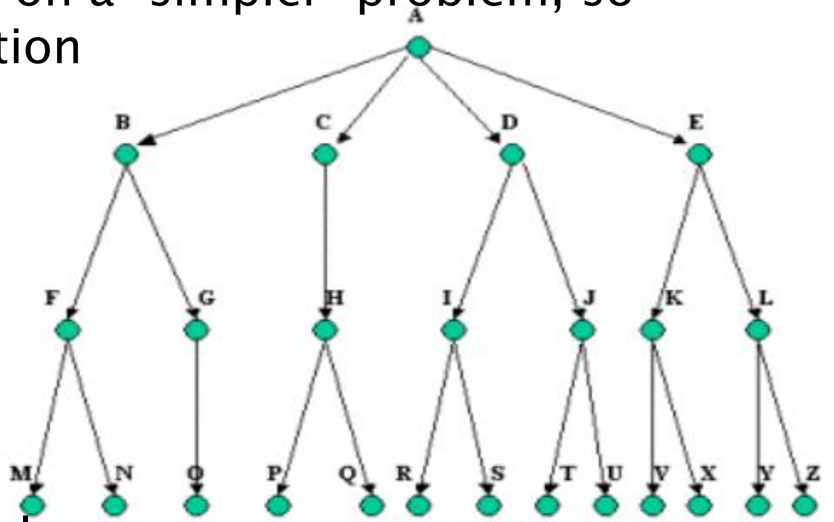
Checkout *Recursion2* project from SVN

# Questions?


Bizarro
I have this recurring dream that I'm lying here telling you about a recurring dream about lying here telling you about a recurring dream about...
zzzz
©DAN PIRARO
BIZARRO.COM
3-6-02

# Recursion

- What is a *recursive* method?
- Answer: *A method that calls itself* but on a "simpler" problem, so that it makes progress toward completion
- When to use recursive methods?
  - Implementing a recursive definition
    $n! = n \times (n-1)!$
  - Implementing methods on a recursive data structure, e.g.:
    Size of tree to the right is the sum of sizes of subtrees B, C, D, E, plus 1
  - Any situation where parts of the whole look like mini versions of the whole
    - Folders within folders on computers
    - Trees




- Pros: easy to implement, easy to understand code, easy to prove code correct
- Cons: takes more space than equivalent iteration (because of function calls)

# Key Rules to Using Recursion

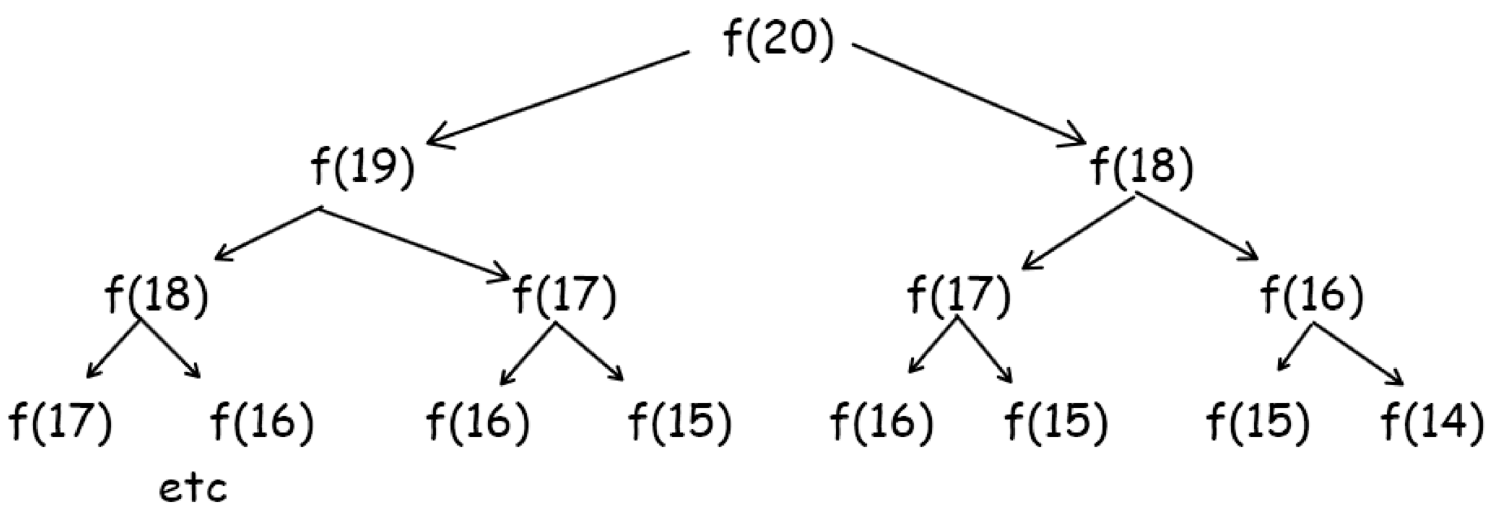
- Always have a **base case** that **doesn't recurse**
- Make sure recursive case always makes **progress**, by **solving a smaller problem**
- **You gotta believe**
  - Trust in the recursive solution
  - Just consider one step at a time

# What the Fib?

- The nth Fibonacci number F(n) is defined by:

  $F(n) = F(n-1) + F(n-2)$ for $n > 1$

  $F(1) = F(2) = 1$
- Why does recursive Fibonacci take so long?!?
  - Hint: How deep is the right-most branch of the tree below? Hence how big the tree? Hence how long does the computation take?
- How can we fix it?
  - Use a memory table! Same idea as what some of you noticed about Ackermann, but more powerful with Fibonacci.




Q1-2

# Memory tables

- To speed up the recursive calculation of the nth Fibonacci number, just:
  1. "Memorize" every solution we find to subproblems, and
  2. Before you recursively compute a solution to a subproblem, look it up in the "memory table"
     - So to compute the nth Fibonacci number, construct an array that has n+1 elements, all initialized to 0. Then call Fib(n).
     - The base case for Fib(k) remains the same as in the naive solution.
     - At the beginning of the recursive step computing Fib(k), see if the $k^{th}$ entry in the array is 0.
       - If it is NOT 0, return it.
       - If it IS 0, compute Fib(k) recursively. Then store the computed value in the $k^{th}$ spot of the array. Then return the computed value.

This is a classic ***time-space tradeoff***
- A deep discovery of computer science
- Studied by "Complexity Theorists"
- Used everyday by software engineers

Tune the solution by varying the amount of storage space used and the amount of computation performed

Q3

# Mutual Recursion

- Two or more methods that call each other repeatedly
  - For example, Hofstadter Female and Male Sequences

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \\ n - M(F(n-1)) & \text{if } n > 0 \end{cases}$$

$$M(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - F(M(n-1)) & \text{if } n > 0 \end{cases}$$

  - Burning Questions for you to figure out **now** by coding:
    - How often are the sequences different in the first 50 positions? first 500? first 5,000? first 5,000,000?

Q4

# Two Mirrors


If you actually do this, what really happens is Douglas Hofstadter appears and talks to you for eight hours about strange loops.

# What is sorting?

- Let’s see…

# Why study sorting?

- Shlemiel the Painter

# What makes a program "good"?

- Correct – meets specifications
- Easy to understand, modify, write
- Uses reasonable set of resources
  - Time (runs fast)
  - Space (main memory)
  - Hard-drive space
  - Peripherals
  - …
- Here we focus on "runs fast" – **how much CPU time does the program / algorithm / problem take?**
  - Others are important too!

# Course Goals for Sorting: You should…

- Be able to describe basic sorting algorithms:
  - Selection sort
  - Insertion sort
  - Merge sort
  - Quicksort
- Know the run-time efficiency of each
- Know the best and worst case inputs for each

# Selection Sort

- Basic idea:
  - Think of the list as having a sorted part (at the beginning) and an unsorted part (the rest)
  - Find the smallest number in the unsorted part
  - Move it to the end of the sorted part (making the sorted part bigger and the unsorted part smaller)

Repeat until unsorted part is empty

# Profiling Selection Sort

- Profiling: collecting data on the run-time behavior of an algorithm

- How long does selection sort take on:
  - 10,000 elements?
  - 20,000 elements?
  - …
  - 100,000 elements?

Q5-6

# Big-Oh motivation: why profiling is not enough

- Results from profiling depend on:
  - Power of machine you use
    - CPU, RAM, etc
  - Operating system of machine you use
  - State of machine you use
    - What else is running? How much RAM is available? …
  - What inputs do you choose to run?
    - Size of input
    - Specific input

# Big-Oh motivation: what it provides

- Big-Oh is a mathematical definition that allows us to:
  - Determine how fast a program is (in big-Oh terms)
  - Share results with others in terms that are universally understood
- Features of big-Oh
  - Allows paper-and-pencil analysis
  - Is much easier / faster than profiling
  - Is a function of the *size of the input*
  - Focuses our attention on *big* inputs
  - Is machine independent

# Analyzing Selection Sort

- Analyzing: calculating the performance of an algorithm by studying how it works, typically mathematically
- Typically we want the relative performance as a function of input size

- Example: For an array of length *n*, how many times does **`selectionSort()`** call **`compareTo()`**?

| Handy Fact |
| --- |
| $1 + 2 + \ldots + (n-1) + n = \frac{n(n+1)}{2}$ |

Q7–12

# Asymptotic analysis

- We care most what happens when **n** (the size of a problem) gets large
  - Is the function basically linear,  quadratic, exponential, etc. ?
  - Consider:  Why do we care most about large inputs?

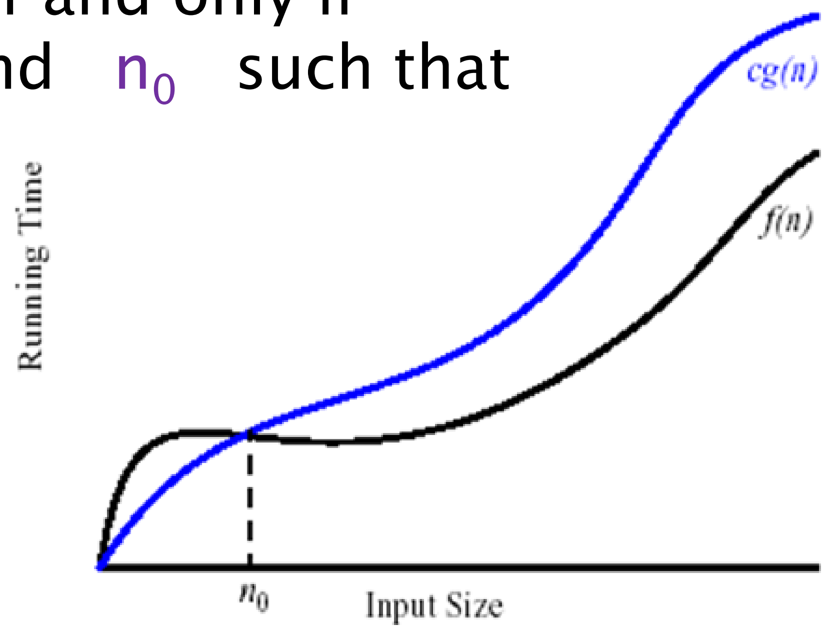
- For example, when **n** is large (or even moderate):
  - The difference between $n^2$ and $n^2 - 3$ is negligible.
  - $n^3$ is pretty large but $2^n$ is REALLY large.

- We say, "selection sort takes on the order of $n^2$ steps"

- Big-Oh gives a formal definition for
  "on the order of"

# Definition of big-Oh

- Formal:
  - We say that $f(n)$ is $O(g(n))$ if and only if
  - there exist constants $c$ and $n_0$ such that
  - for every $n \geq n_0$ we have
  - $f(n) \leq c \times g(n)$




- Informal:
  - $f(n)$ is *roughly proportional* to $g(n)$, for large $n$
- Example: $7n^3 + 24n^2 + 3000n + 45$ is $O(n^3)$
  - Because it is $\leq 3{,}077 \times n^3$ for all $n \geq 1$