CSCI 136
Data Structures &
Advanced Programming

Lecture 8

Fall 2018

Instructors: Bills

Administrative Details

- Lab 3 Wednesday!
  - You *may* work with a partner
  - Come to lab with a plan!
  - Try to answer questions before lab

Last Time

- Vector Implementation
- Miscellany: Wrappers
- Condition Checking
  - Pre- and post-conditions, Assertions
- Asymptotic Growth & Measuring Complexity

Today

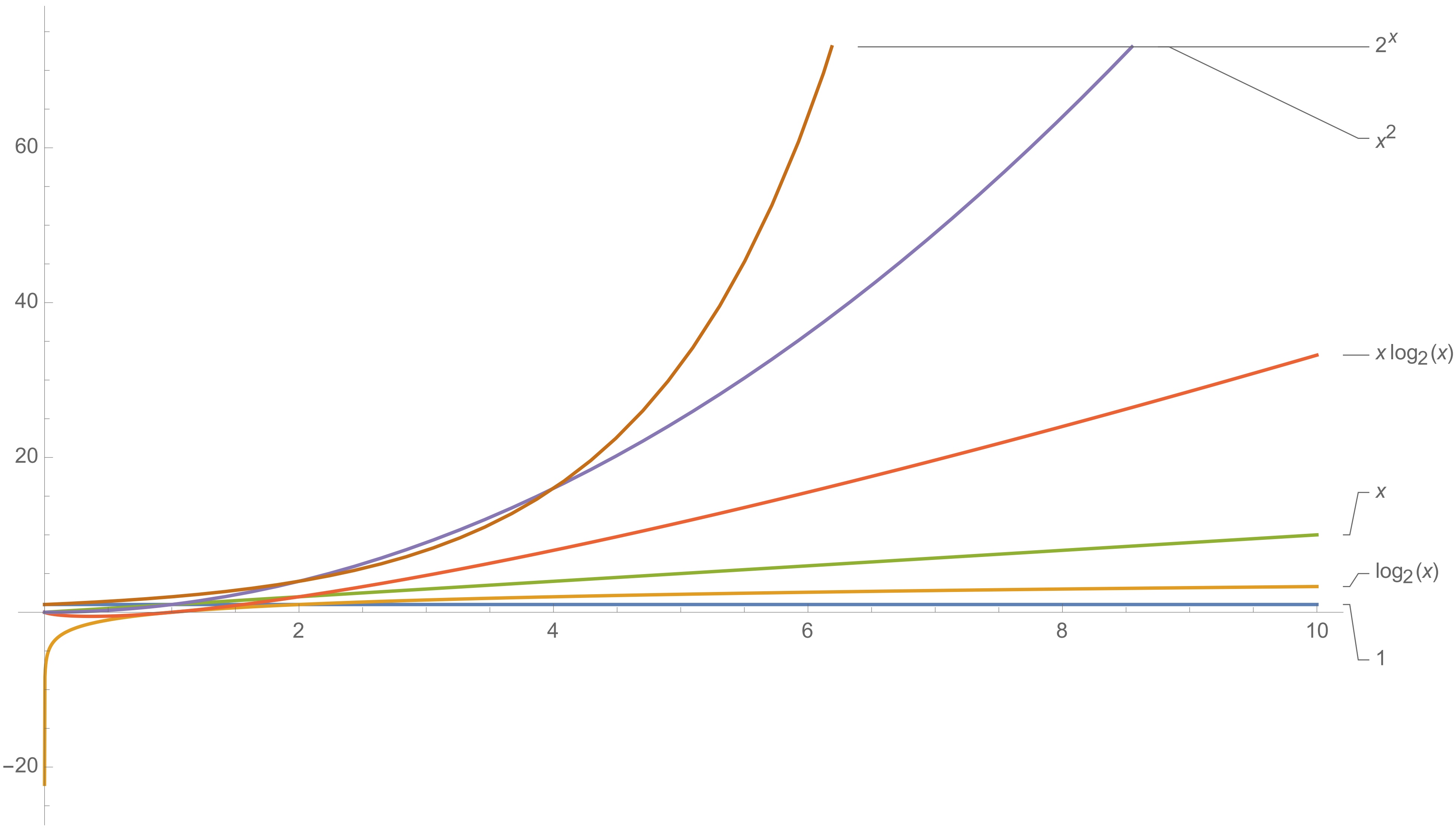
- Measuring Growth
  - Big-O
- Introduction to Recursion & Induction

Function Growth

Consider the following functions, for $x \geq 1$

- $f(x) = 1$
- $g(x) = \log_2(x)$ // Reminder: if x=2^n, $\log_2(x) = n$
- $h(x) = x$
- $m(x) = x \log_2(x)$
- $n(x) = x^2$
- $p(x) = x^3$
- $r(x) = 2^x$

Function Growth



Function Growth & Big-O

- Rule of thumb: ignore multiplicative constants
- Examples:
  - Treat n and n/2 as same order of magnitude
  - $n^2/1000$, $2n^2$, and $1000n^2$ are "pretty much" just n^2
  - $a_0n^k + a_1n^{k-1} + a_2n^{k-2} + \cdots a_k$ is roughly n^k
- The key is to find the most *significant* or *dominant* term
- Ex: $\lim_{x\to\infty} (3x^4 - 10x^3 - 1)/x^4 = 3$ (Why?)
  - So $3x^4 - 10x^3 - 1$ grows "like" x^4

Asymptotic Bounds (Big-O Analysis)

- A function f(n) is *O(g(n))* if and only if there exist positive constants c and n_0 such that

$$|f(n)| \leq c \cdot g(n) \text{ for all } n \geq n_0$$

- g is "at least as big as" f **for large n**
  - Up to a multaplicative constant c!
- Example:
  - $f(n) = n^2/2$ is $O(n^2)$
  - $f(n) = 1000n^3$ is $O(n^3)$
  - $f(n) = n/2$ is $O(n)$

Determining "Best" Upper Bounds

- We typically want the *most conservative* upper bound when we estimate running time
  - And among those, the *simplest*
- Example: Let $f(n) = 3n^2$
  - $f(n)$ is $O(n^2)$
  - $f(n)$ is $O(n^3)$
  - $f(n)$ is $O(2^n)$ (see next slide)
  - $f(n)$ is NOT $O(n)$ (!!)
- "Best" upper bound is $O(n^2)$
- We care about **c** and $\mathbf{n_0}$ in practice, but focus on size of **g** when designing algorithms and data structures

What's n_0? Messy Functions

- Example: Let $f(n) = 3n^2 - 4n + 1$. f(n) is $O(n^2)$
  - Well, $3n^2 - 4n + 1 \leq 3n^2 + 1 \leq 4n^2$, for $n \geq 1$
  - So, for c = 4 and $n_0 = 1$, we satisfy Big-O definition
- Example: Let $f(n) = n^k$, for any fixed $k \geq 1$. **f(n) is** $O(2^n)$
  - Harder to show: Is $n^k \leq c\, 2^n$ for some c > 0 and large enough n?
  - It is if and only if $\log_2(n^k) \leq \log_2(2^n)$, that is, iff $k \log_2(n) \leq n$.
  - That is iff $k \leq n/\log_2(n)$. But $n/\log_2(n) \rightarrow \infty$ as $n \rightarrow \infty$
  - This implies that for some n_0 on $n/\log_2(n) \geq k$ if $n \geq n_0$
  - Thus $n \geq k \log_2(n)$ for $n \geq n_0$ and so $2^n \geq n^k$

Vector Operations : Worst-Case

For n = Vector size (*not* capacity!):

- O(1): size(), capacity(), isEmpty(), get(i), set(i), firstElement(), lastElement()
- O(n): indexOf(), contains(), remove(elt), remove(i)
- What about add methods?
  - If Vector doesn't need to grow
    - add(elt) is O(1) but add(elt, i) is O(n)
  - Otherwise, depends on ensureCapacity() time
    - Time to compute newLength : O($\log_2(n)$)
      - Assuming doubling rule!
    - Time to copy array: O(n)
    - $O(\log_2(n)) + O(n)$ is O(n)

Vectors: Add Method Complexity

Suppose we grow the Vector's array by a fixed amount d.

How long does it take to add n items to an empty Vector?

- The array will be copied each time its capacity needs to exceed a multiple of d
  - At sizes 0, d, 2d, … , n/d.
- Copying an array of size kd takes ckd steps for some constant c, giving a total of

$$\sum_{k=1}^{n/d} ckd = cd \sum_{k=1}^{n/d} k = cd\,(\frac{n}{d})(\frac{n}{d}+1)/2 = O(n^2)$$

Vectors: Add Method Complexity

Suppose we grow the Vector's array by doubling.

How long does it take to add n items to an empty Vector?

- The array will be copied each time its capacity needs to exceed a power of 2
  - At sizes 0, 1, 2, 4, 8 … $2^{\log_2 n}$
- Copying an array of size 2^k takes c 2^k steps for some constant c, giving a total of

$$\sum_{k=1}^{\log_2 n} c2^k = c\ \sum_{k=1}^{\log_2 n} 2^k = c\ (2^{\log_2 n+1} - 1) = O(n)$$

- Very cool!

Common Complexities

For n = measure of problem size:

- $O(1)$: constant time and space
- $O(\log n)$: divide and conquer algorithms, binary search
- $O(n)$: linear dependence, simple list lookup
- $O(n \log n)$: divide and conquer sorting algorithms
- $O(n^2)$: matrix addition, selection sort
- $O(n^3)$: matrix multiplication
- $O(n^k)$: cell phone switching algorithms
- $O(2^n)$: subset sum, graph 3-coloring, satisfiability, ...
- $O(n!)$: traveling salesman problem (in fact $O(n^2 2^n)$)

Recursion

- General problem solving strategy
  - Break problem into smaller pieces
  - Sub-problems may look a lot like original – are often smaller versions of same problem

Recursion

- Many algorithms are recursive
  - Can be easier to understand (and prove correctness/state efficiency of) than iterative versions
- Today we will review recursion and then talk about techniques for reasoning about recursive algorithms

Factorial

- n! = n • (n-1) • (n-2) • … • 1
- How can we implement this?
  - We could use a for loop…

```
int product = 1;
for(int i = 1;i <= n; i++)
        product *= i;
```

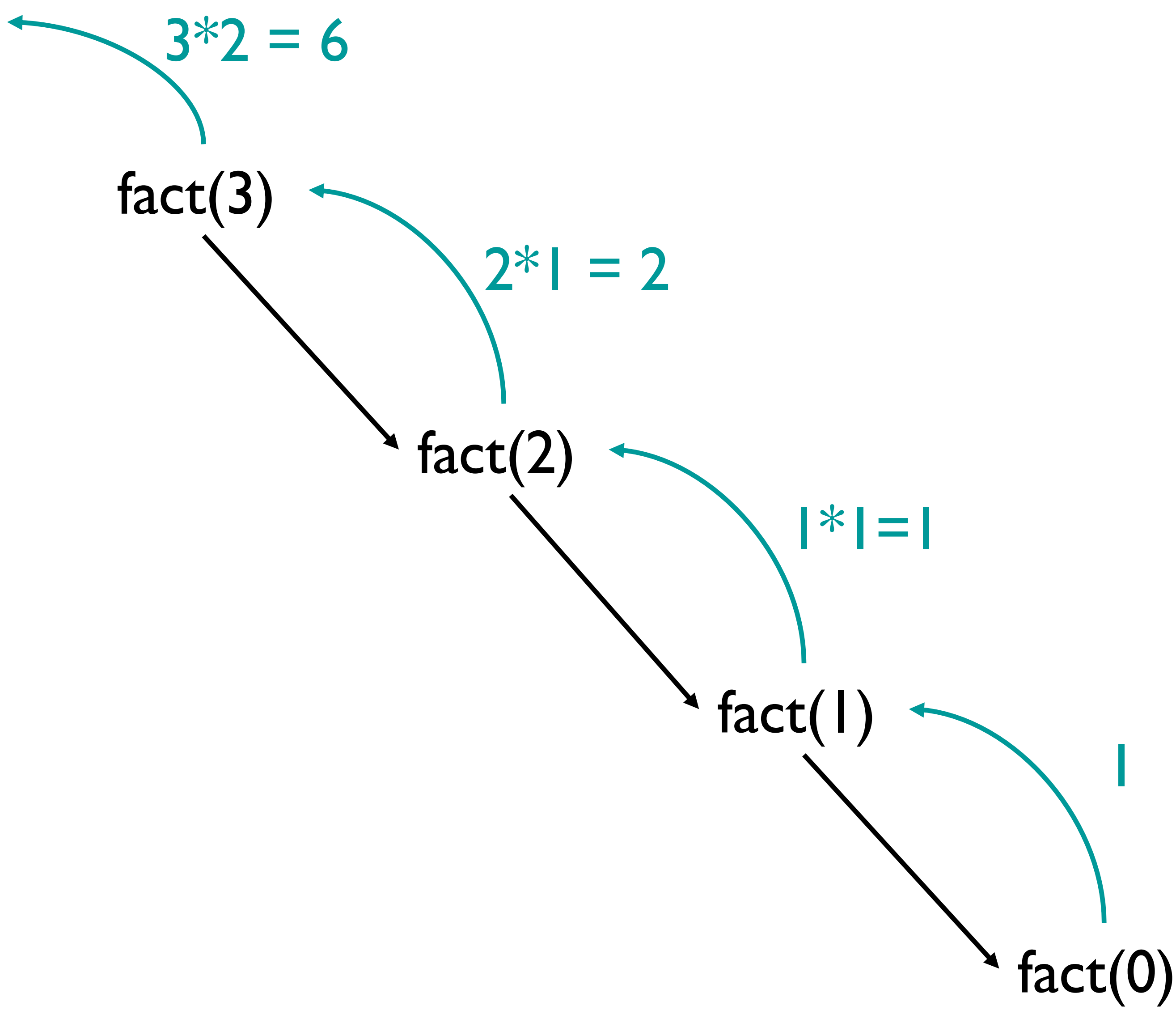
- But we could also write it recursively….

Factorial

- n! = n • (n-1) • (n-2) • … • 1
- But we could also write it recursively
  - n! = n • (n-1)!
  - 0! = 1

```
// Pre: n >= 0
public static int fact(int n) {
      if (n==0) return 1;
      else return n*fact(n-1);
}
```

Factorial



Factorial

- In recursion, we always use the same basic approach
- What's our base case? [Sometimes "cases"]
  - $n=0$; $fact(0) = 1$
- What's the recursive relationship?
  - $n>0$; $fact(n) = n \bullet fact(n-1)$

Fibonacci Numbers

- 1, 1, 2, 3, 5, 8, 13, ...
- Definition
  - $F_0 = 1$, $F_1 = 1$
  - For $n > 1$, $F_n = F_{n-1} + F_{n-2}$
- Inherently recursive!
- It appears almost everywhere
  - Growth: Populations, plant features
  - Architecture
  - Data Structures!

fib.java

```
public class fib{
   // pre: n is non-negative
    public static int fib(int n) {
       if (n==0 || n == 1) {
          return 1;
       }
       else {
          return fib(n – 1) + fib(n – 2);
       }
    }

    public static void main(String args[]) {
       System.out.println(fib(Integer.valueOf(args[0]).intValue()));
    }

}
```

Towers of Hanoi

- Demo
- Base case:
  - One disk: Move from start to finish
- Recursive case (n disks):
  - Move smallest n-1 disks from start to temp
  - Move bottom disk from start to finish
  - Move smallest n-1 disks from temp to finish
- Let's try to write it....

Longest Increasing Subsequence

- Given an array a[] of positive integers, find the largest subsequence of (not necessary consecutive) elements such that for any pair a[i], a[j] in the subsequence, if $i<j$, then $a[i] < a[j]$.
- Example 10 7 12 3 5 11 8 9 1 15 has 3 5 8 9 15 as its longest increasing subsequence (LIS).
- How could we find an LIS of a[]?
- How could we prove our method was correct?
- Let's think....

Longest Increasing Subsequence

- (Brilliant) Observation: A LIS for a[1 ... n] either contains a[1] … or it doesn't.
- Therefore, a LIS for a[1 ... n] either
  - contains a[1] along with an LIS for a[2 ... n] such that every element in the LIS is > a[1], or
  - Is a LIS for a[2 ... n]
- How could we find a LIS of a[]?
  - Use the B.O. to build a recursive method
- How could we prove our method was correct?
  - Induction!

Longest Increasing Subsequence

```
// Pre: curr <= length
public static int lisHelper(int[] arr, int curr, int maxSoFar ) {
    if(curr == arr.length) return 0;
    if(arr[curr] <=  maxSoFar)
        return lisHelper(arr, curr +1,maxSoFar);
    else
        return Math.max(
                lisHelper(arr,curr +1,maxSoFar),
                1 + lisHelper(arr, curr +1, arr[curr]));
  }
```

Recursion Tradeoffs

- Advantages
  - Often easier to construct recursive solution
  - Code is usually cleaner
  - Some problems do not have obvious non-recursive solutions
- Disadvantages
  - Overhead of recursive calls
  - Can use lots of memory (need to store state for each recursive call until base case is reached)
    - E.g. recursive fibonacci method