CSCI 136
Data Structures & Advanced Programming

Bill Jannen

Lecture 8

Feb 22, 2017

Announcements

- Lab 1 returned
- Lab 3 sections start today
  - Questions about warm-up?
- Next few lectures: Jon!

Last Time

- Finished implementing Vector.java
- Talked about Big-O analysis

Today’s Outline

- More on Big-O analysis
- Recursion
- Induction

Big-O Analysis

- A *general tool* for understanding how our resource consumption changes as the size of our inputs increase
  - Time
  - Space
- We care about trends
  - Rule of thumb: ignore constants
  - Consider the dominant term

Asymptotic Bounds (Big-O Analysis)

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

  $$|f(n)| \leq c * g(n) \text{ for all } n \geq n_0$$
- "g" is bigger than "f" **for large n**

- Consider 2^n and n^2 for $0 <= n <= 4$
  - Which is larger?
- What about $n > 4$?

Careful Counting

- What is the Big-O cost of the following code:

```
public static Vector<Integer> descendingVector(int n) {
   Vector v = new Vector(n); // can add n items before need to grow
   for (int i = 0; i < n; i++) {
        v.add(n-i);
   }
   return v;
}
```

v.add(i) is O(1) unless we must grow the array

Call v.add(i) n times

O(n)

Careful Counting

- What is the Big-O cost of the following code:

```
public static Vector<Integer> descendingVector(int n) {
   Vector v = new Vector(n); // can add n items before need to grow
   for (int i = 0; i < n; i++) {
        v.add(0, i);
   }
   return v;
}
```

Only call v.add() n times, but each requires shifting i elements.

$O(n^2)$

Moving on…

Recursion

- General problem solving strategy
  - Base case
    - The smallest, often simplest, version of a problem.
    - Where our code “bottoms out”
  - Inductive leap
    - We assume we have a solution to a smaller version of our problem, and we solve our current version of the problem using that solution.

Recursion is Beautiful

- Many algorithms are recursive
  - Often 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 x (n-1) x (n-2) x … x 1
- How can we implement this?
  - We could use a while loop…

- But we could also write it recursively
  - n! = n x (n-1)!

Factorial

- In recursion, we always use the same basic approach
- What's our base case?
  - n=0; fact(0) = 1
- What's our recursive case?
  - n>0; fact(n) = n x fact(n-1)

fact.java

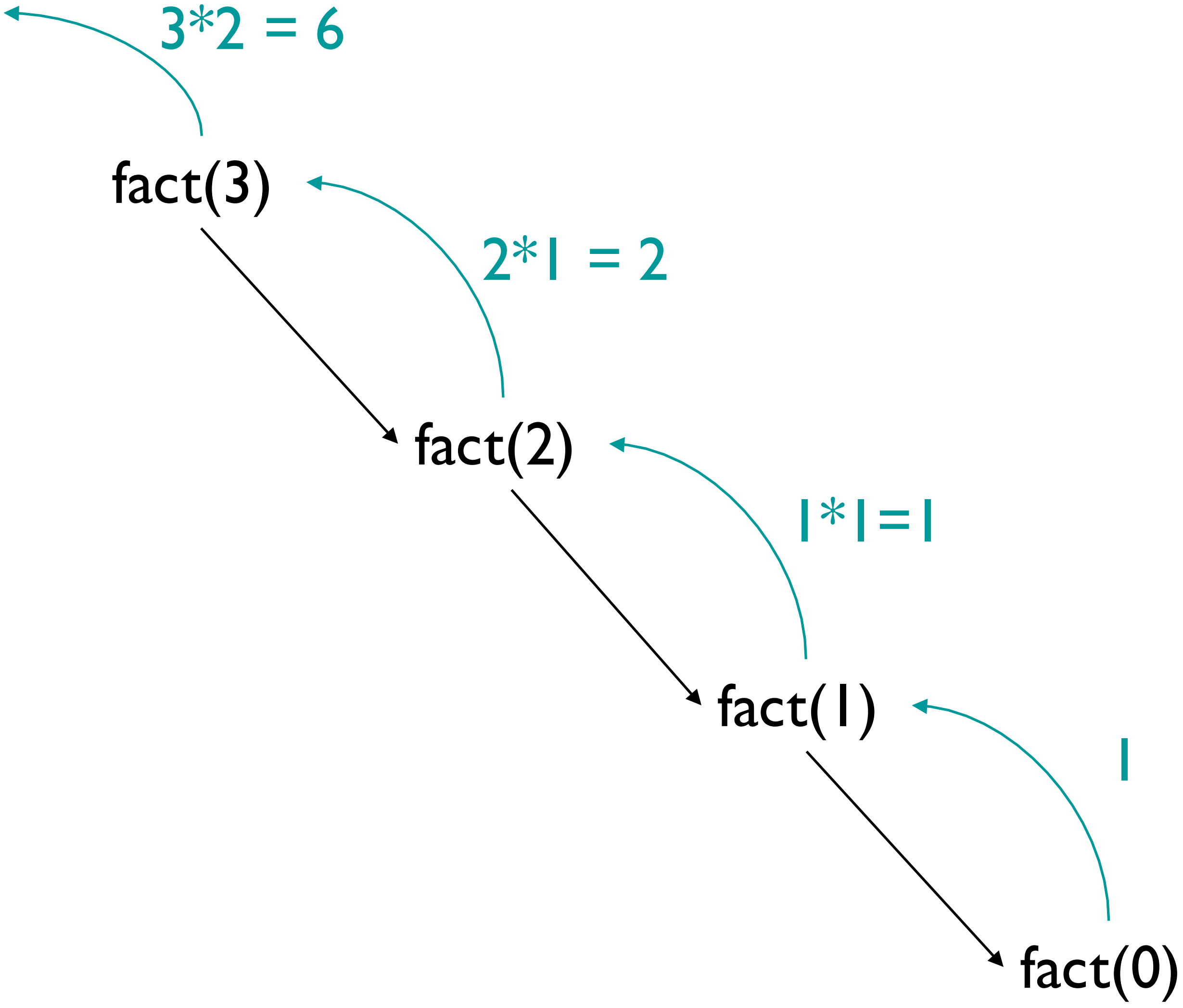
```
public class fact{

    public static int fact(int n) {
       if (n==0) {
          return 1;
       }
       else {
          return n*fact(n-1);
       }
    }

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

}
```

Factorial



3*2 = 6

fact(3)

2*1 = 2

fact(2)

1*1=1

fact(1)

1

fact(0)

Mathematical Induction

- The mathematical equivalent of recursion is induction
- Induction is a proof technique

1. Prove all necessary base cases
2. State that the assumption holds for all values from the base case up to (but not including) the nth case.
3. Prove that, using the simpler cases, the nth case holds.
4. Claim that by induction on n, it is true for all cases more complicated than the nth case

Mathematical Induction

- Examples

$$P = \sum_{i=0}^{n} i = 0 + 1 + \ldots + n = \frac{n(n+1)}{2}$$

- Proof by induction:
  - Base case: P is true for 0
  - Inductive hypothesis: If P is true for all k<n, then P is true for n.
  - P is true for n using the inductive hypothesis.

$$P = \sum_{i=0}^{n} i = 0 + 1 + \ldots + n = \frac{n(n+1)}{2}$$

- Base case: P is true for 0

$$0 = \frac{0(0+1)}{2}$$

- Inductive hypothesis: P is true for all k<n.
- Show P is true for n using the inductive hypothesis.

$$0 + 1 + 2 + \ldots + (n-1) + n$$

$$[0 + 1 + 2 + \ldots + (n-1)] + n$$

$$\left[\frac{(n-1)((n-1)+1)}{2}\right] + n$$

$$\frac{n^2+n}{2} + \frac{2n}{2}$$

$$\frac{n^2+n}{2}$$

$$\frac{n(n+1)}{2}$$

Use our inductive hypothesis (for n-1)

Induction in CS?

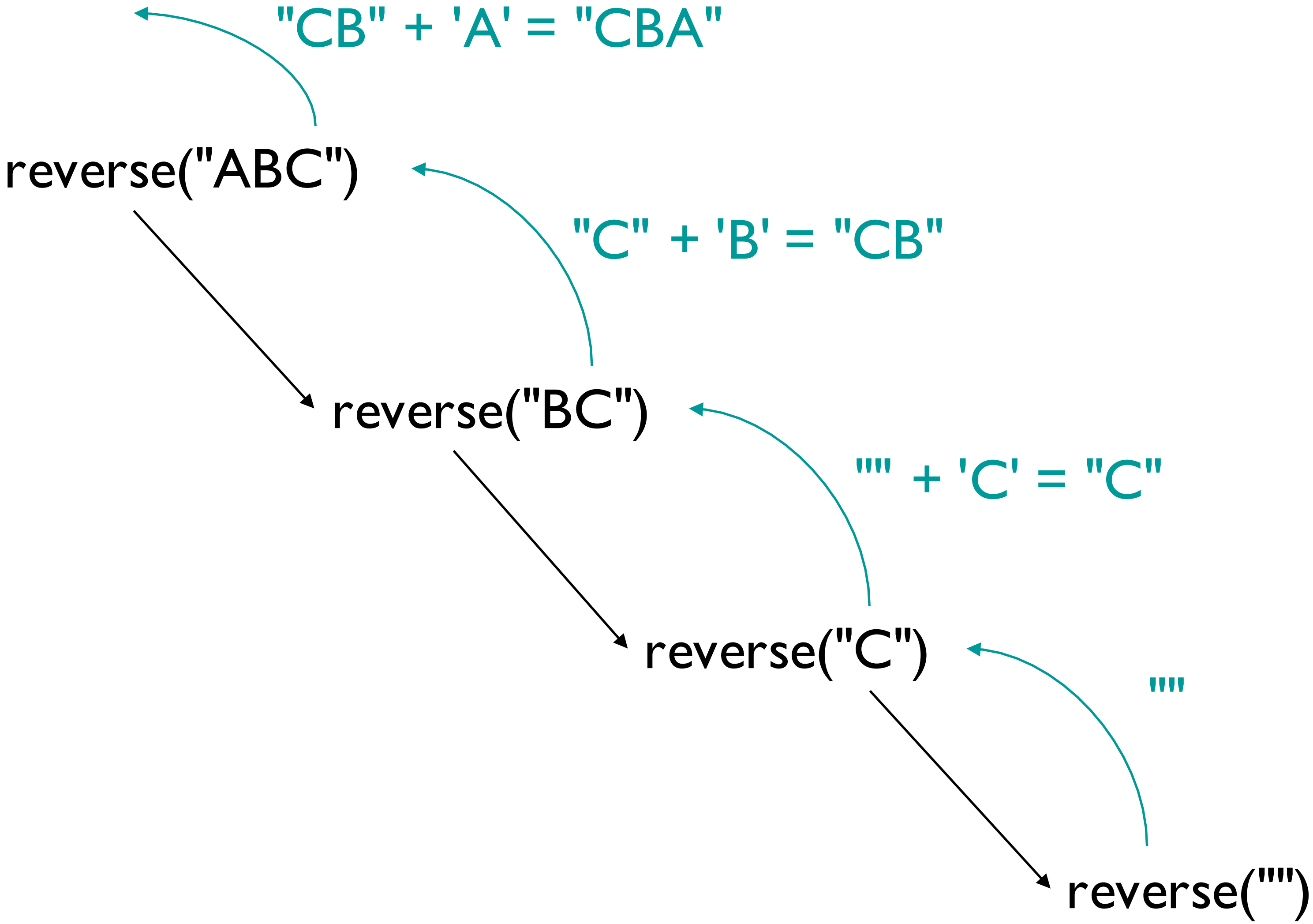
- What does induction have to do with recursion?
  - Same form!
    - Base case
    - Inductive case that uses simpler form of problem
- Example: factorial
  - Prove that fact(n) requires n multiplications
    - Base case: n = 0 returns 1, 0 multiplications
    - Assume true for all k<n, so fact(k) requires k multiplications.
    - fact(n) performs one multiplication (n*fact(n-1)). We know that fact(n-1) requires n-1 multiplications. 1+n-1 = n, therefore fact(n) requires n multiplications.

Problem Solving

- Write a function that takes a String as input and returns a new String where the characters are in reverse order.
- Write the Vector.add(int index, E element) method as a recursive function.

What is your base case?
What is your inductive leap?

Visualizing Reverse



"CB" + 'A' = "CBA"

reverse("ABC")

"C" + 'B' = "CB"

reverse("BC")

"" + 'C' = "C"

reverse("C")

""

reverse("")

Mathematical Induction

- Prove: $\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$

- Prove: $0^3 + 1^3 + \ldots + n^3 = (0 + 1 + \ldots + n)^2$

Lab Warm Up Problems

- Digit Sum
  - public static int digitSum(int n)
  - Base case?
  - Recursive case?

- Subset Sum
  - public static boolean canMakeSum(int set[], int target)
  - Helper:
    - public static boolean canMakeSumHelper(int set[], int target, int index)
  - Base case?
  - Recursive case?

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)