CSCI 136
Data Structures & Advanced Programming

Lecture 10

Fall 2017

Instructors: Bills

Administrative Details

- First Problem Set is online
- Due by 11:00 pm Thursday night
  - Drop it off in your instructor's CS mailbox outside of TCL 303
  - If next Friday is NOT Mountain Day, you can bring it to class instead!

Last Time

- Measuring Growth
  - Big-O

Today

- Applying O() to Compute Complexity
- Recursion
- Mathematical Induction (Weak)
- Recursion on Chains
- Mathematical Induction (Strong)

Input-dependent Running Times

- Algorithms may have different running times for different inputs of a given size
- Best case (typically not useful)
  - Find item in first place that we look O(1)
- Worst case (generally useful, sometimes misleading)
  - Don't find item in list O(n)
- Average case (useful, but often hard to compute)
  - Linear search O(n)

Vectors vs. SLL

- Compare runtime of
  - size
  - addLast, removeLast, getLast
  - addFirst, removeFirst, getFirst
  - get(int index), set(E d, int index)
  - remove(int index)
  - contains(E d)
  - remove(E d)

List Operations : Worst-Case

For a singly-linked list of n items

- O(1): size(), isEmpty(), firstElement()
  - lastElement() (if the list has a tail reference)
- O(n): get(i), set(i), indexOf(), contains(), remove(elt), remove(i)
  - lastElement() (if the list *doesn't* have a tail reference)
- What about add/remove methods?
  - O(1): addFirst(), removeFirst()
  - O(n): add(i),add()/addLast(), remove(i)/remove()/removeLast()

For a doubly-linked list, adding/removing from the tail becomes O(1)

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($\log_2(n)$)
      - If doubling; otherwise could be O(n) : n is new array size
    - 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

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

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!

Vectors vs. SLL

Operation	Vector	SLL
size	O(1)	O(1)
addLast	O(1) or O(n)(if resize)	O(n)
removeLast	O(1)	O(n)
getLast	O(1)	O(n)
addFirst	O(n)	O(1)
removeFirst	O(n)	O(1)
getFirst	O(1)	O(1)
get(i)	O(1)	O(n)
set(i)	O(1)	O(n)
remove(i)	O(n)	O(n)
contains	O(n)	O(n)
remove(o)	O(n)	O(n)

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 sub-problems of same type
  - Solve sub-problems
  - Combine sub-problem solutions into solution for original problem

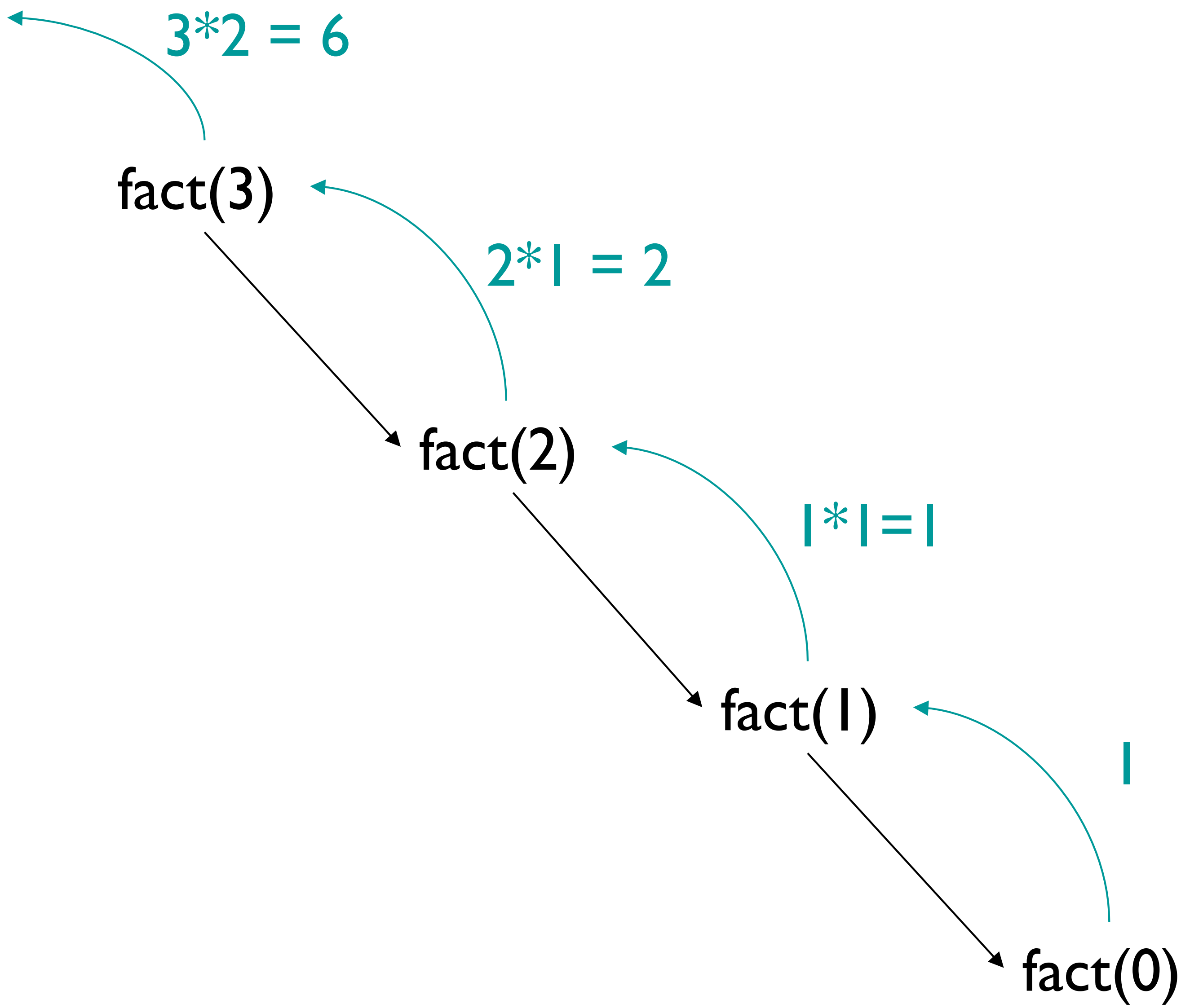
Recursion

- 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

- $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$
- How can we implement this?
  - We could use a for loop…
- But we could also write it recursively
  - $n! = n \cdot (n-1)!$
  - $0! = 1$

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 · fact(n-1)

fact.java

```
public class fact{

    // Pre: n >= 0
    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()));
    }

}
```

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....

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

Alternate contains() for Vector

```
// Helper method: returns true if elt has index in range from..to
public boolean contains(E elt, int from, int to) {
   if (from > to)
        return false; // Base case: empty range
   else
        return elt.equals(elementData[from]) ||
                contains(elt, from+1, to);
}

public boolean contains(E elt) {
   return contains(elt, 0, size()-1); }
```

- What's the time complexity of contains?
  - O(to – from + 1) = O(n) (n is the portion of the array searched)
  - Why?
    - Bootstrapping argument! True for: to – from = 0, to – from = 1, …
- Let's formalize this bootstrapping idea....

Mathematical Induction

- The mathematical cousin of recursion is induction
- Induction is a proof technique
- Reflects the structure of the natural numbers
- Use to simultaneously prove an infinite number of theorems!

Mathematical Induction

- Example: Prove that for every $n \geq 0$

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

- Proof by induction:
  - Base case: P_n is true for $n = 0$ (just check it!)
  - Inductive hypothesis: If P_n is true for some $n \geq 0$, then P_{n+1} is true.

$$P_{n+1}: 0 + 1 + \ldots + n + (n+1) = \frac{(n+1)\big((n+1)+1\big)}{2} = \frac{(n+1)(n+2)}{2}$$

Check: $0 + 1 + \ldots + n + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$

- First equality holds by assumed truth of P_n!

Mathematical Induction

Principle of Mathematical Induction (Weak)

Let P(0), P(1), P(2), ... Be a sequence of statements, each of which could be either true or false. Suppose that

1. P(0) is true, and
2. For all $n \geq 0$, if P(n) is true, then so is P(n+1).

Then all of the statements are true!

Note: Often Property 2 is stated as

2. For all $n > 0$, if P(n-1) is true, then so is P(n).

   Apology: I do this a lot, as you'll see on future slides!