CSCI 136
Data Structures &
Advanced Programming

Lecture 8

Fall 2019

Instructors: Bill & Sam

Administrative Details

- Remember: First Problem Set is online
- Due at beginning of class on Friday
- Lab 3
  - You *may* work with a partner
    - Fill out the Google Form by 4 pm today (check email!)
  - Come to lab with a plan!
  - Answer questions before lab
  - Released later today
- Lab 1 grades later today

Last Time

- Measuring Computational Complexity

Today

- More Recursion
- Mathematical Induction (Weak)

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 : $O(\log n)$
    - Time to copy array: $O(n)$
    - $O(\log n) + O(n)$ is $O(n)$

Vector: 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
- Copying an array of size kd takes ckd steps for some constant c, giving a total of

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

Vector: Add Method Complexity

Suppose we want to 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 it's capacity needs to exceed a power of 2.
  - At sizes $0, 1, 2, 4, 8, \ldots, 2^{\lfloor \log_2 n \rfloor}$
- Copying an array of size 2^k takes $c2^k$ steps for some constant c, giving a total of:

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

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
- $O(n^{12})$: Original AKS primality test for n-bit integers
- $O(2^n)$: 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)
  - Sub-problems are typically smaller versions of same problem

Recursion

- Many algorithms are recursive
  - Can be easier to understand, prove correctness, or determine efficiency
- Today we will review recursion and then talk about techniques for reasoning about recursive algorithms

Factorial

- $n! = n(n-1)(n-2)(n-3)\ldots(3)(2)(1)$
- How can we implement this?
  - We could use a for loop…

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

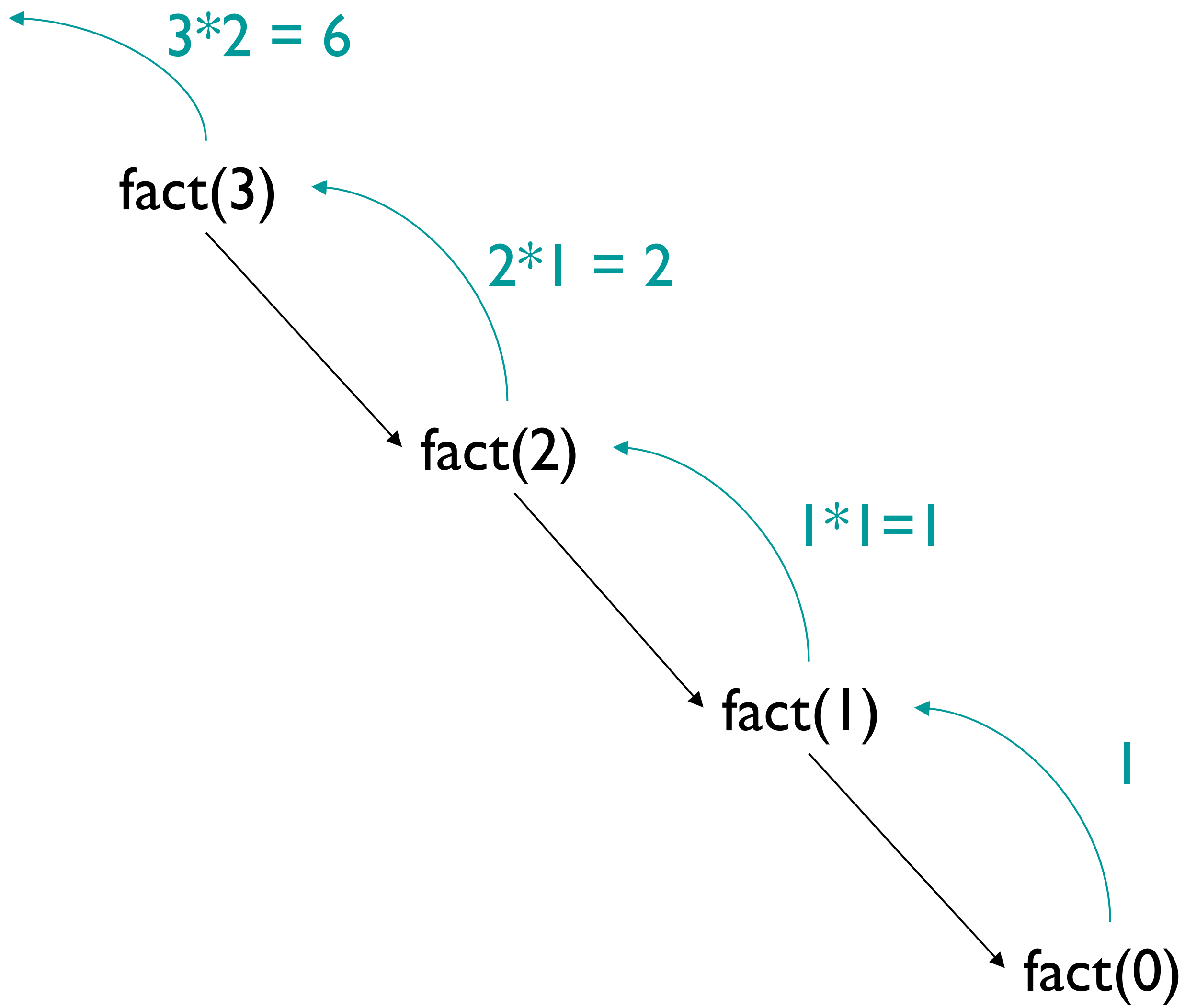
- But we could also write it recursively….

Factorial

- $n! = n(n-1)(n-2)(n-3)\ldots(3)(2)(1)$
- Recursive definition (what "…" really means!)
  - $n! = n * (n-1)!$
  - $0! = 1$

```
// 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 * 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()));
    }

}
```

Demo: RecursiveMethods.java….

Question: Why is fib so slow?!

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 ≥ 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!)
  - Induction step: If P_n is true for some n≥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!

Mathematical Induction

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

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

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

Note: I'm doing the n-1 → n version

$$\begin{aligned}
(0^3 + 1^3 + \ldots n^3) &= (0^3 + 1^3 + \ldots + (n-1)^3) + n^3 \\
\text{Induction☞} &= (0 + 1 + \ldots + (n-1))^2 + n^3 \\
&= \left(\frac{n(n-1)}{2}\right)^2 + n^3 \\
&= n^2\left(\frac{(n-1)^2 + 4n}{4}\right) \\
&= n^2\left(\frac{n^2 + 2n + 1}{4}\right) \\
&= n^2\left(\frac{(n+1)^2}{4}\right) \\
&= \left(\frac{n(n+1)}{2}\right)^2 \\
&= (0 + 1 + \ldots + n)^2
\end{aligned}$$

What about Recursion?

- 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.

Counting Method Calls

- Example: Fibonacci
  - Prove that fib(n) makes at least fib(n) calls to fib()
    - Base cases: n = 0: 1 call; n = 1; 1 call
    - Assume that for some $n \geq 2$, fib(n-1) makes at least fib(n-1) calls to fib() and fib(n-2) makes at least fib(n-2) calls to fib().
    - Claim: Then fib(n) makes at least fib(n) calls to fib()
      - 1 initial call: fib(n)
      - By induction: At least fib(n-1) calls for fib(n-1)
      - And as least fib(n-2) calls for fib(n-2)
      - Total: $1 + \text{fib}(n-1) + \text{fib}(n-2) \geq \text{fib}(n-1) + \text{fib}(n-2) = \text{fib}(n)$ calls
    - Note: Need two base cases!
  - One can show by induction that for n > 10: fib(n) > $(1.5)^n$
  - Thus the number of calls grows exponentially!
  - We can visualize this with a *method call graph*….

Mathematical Induction : Version 2

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 and P_1 are true, and
2. For all $n \geq 2$, if P_{n-1} and P_{n-2} are true, then so is P_n.

Then all of the statements are true!

Other versions:

- Can have k > 2 base cases
- Doesn't need to start at 0

Example: Binary Search

- Given an array a[] of positive integers in increasing order, and an integer x, find location of x in a[].
  - Take “indexOf” approach: return −1 if x is not in a[]

```
protected static int recBinarySearch(int a[], int value,
             int low, int high) {
  if (low > high) return -1;
  else {
      int mid = (low + high) / 2;       //find midpoint
      if (a[mid] == value) return mid;  //first comparison
                                        //second comparison
      else if (a[mid] < value)          //search upper half
      return recBinarySearch(a, value, mid + 1, high);
       else                             //search lower half
            return recBinarySearch(a, value, low, mid - 1);
  }
```

Binary Search takes O(log n) Time

Can we use induction to prove this?

- Claim: If n = high - low $+1$, then recBinSearch performs at most c $(1 + \log n)$ operations, where c is *twice* the number of statements in recBinSearch
- Base case: n = 1: Then low = high so only c statements execute (method runs twice) and $c \leq c(1 + \log 1)$
- Assume that claim holds for some $n \geq 1$, does it hold for $n + 1$? [Note: $n + 1 > 1$, so low < high]
- Problem: Recursive call is *not* on n---it's on n/2.
- Solution: We need a better version of the PMI….

Mathematical Induction

Principle of Mathematical Induction (Strong)

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

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

Then all of the statements are true!

Binary Search takes O(log n) Time

Try again now:

- Assume that for some $n \geq 1$, the claim holds *for all* $k \leq n$, does claim hold for n+1?
- Yes! Either
  - x = a[mid], so a constant number of operations are performed, or
  - RecBinSearch is called on a sub-array of size at most n/2, and by induction, at most c(1 + log (n/2)) operations are performed.
    - This gives a total of at most c + c(1 + log(n/2)) = c + c(log(2) + log(n/2)) = c + c(log n) = c(1 + log n) statements

Longest Increasing Subsequence

- Given an array a[] of positive integers, find the length of 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), so the length is 5.
- How could we find the LIS length of a[]?
- How could we prove our method was correct?
- Let’s think....

Longest Increasing Subsequence

- We'll assume all numbers are positive
- (Brilliant) Observation: A LIS for a[1 ... n] either contains a[1] … or it doesn't.
- Therefore, a LIS for a[1 ... n] either
  - Doesn't contain a[1] and is just a LIS for a[2 ... n]
  - Does contain a[1], along with an LIS for a[2 ... n] such that every element in the LIS is > a[1], or
- So the LIS length is either
  - Or the LIS length for a[2..n]
  - 1 + LIS length for a[2..n] where every element in LIS is > a[1]
- So the problem to solve is: find the LISL for a[] given that every element in LIS is at least some *threshold* value

Longest Increasing Subsequence

```
// Pre: curr < arr.length
// Post: returns length of LIS of arr[curr…] having all > threshold
public static int lisIHelper(int[] arr, int curr, int threshold ) {
    if(curr == arr.length -1)
        if (return arr[curr] > threshold) return 1;
        else return 0;
    else
        int usingFirst = 0;
        if(arr[curr] > threshold)
            usingFirst  = 1 + lisIHelper(arr, curr+1, arr[curr]);
        int notUsingFirst = lisIHelper(arr, curr+1, threshold);
        return Math.max(usingFirst, notUsingFirst);
  }
```

Bubble Sort

- First Pass:
  - (**5** 1 3 2 9) → (1 **5** 3 2 9)
  - (1 **5** 3 2 9) → (1 3 **5** 2 9)
  - (1 3 **5** 2 9) → (1 3 2 **5** 9)
  - (1 3 2 **5** 9) → (1 3 2 5 **9**)
- Second Pass:
  - (**1** 3 2 5 9) → (**1** 3 2 5 9)
  - (1 **3** 2 5 9) → (1 2 **3** 5 9)
  - (1 2 **3** 5 9) → (1 2 3 **5** 9)
- Third Pass:
  - (**1** 2 3 5 9) -> (**1** 2 3 5 9)
  - (1 **2** 3 5 9) -> (1 **2** 3 5 9)
- Fourth Pass:
  - (**1** 2 3 5 9) -> (**1** 2 3 5 9)

http://www.youtube.com/watch?v=lyZQPjUT5B4
http://www.visualgo.net/sorting

Sorting Preview: Insertion Sort

- Simple sorting algorithm that works by building a sorted list one entry at a time
- Less efficient on large lists than more advanced algorithms
- Advantages:
  - Simple to implement and efficient on small lists
  - Efficient on data sets which are already substantially sorted
- Time complexity
  - $O(n^2)$
- Space complexity
  - $O(n)$

Sorting Preview: Insertion Sort

- 5 7 0 3 4 2 6 1
- 5 7 0 3 4 2 6 1
- 0 5 7 3 4 2 6 1
- 0 3 5 7 4 2 6 1
- 0 3 4 5 7 2 6 1
- 0 2 3 4 5 7 6 1
- 0 2 3 4 5 6 7 1
- 0 1 2 3 4 5 6 7

Sorting Preview: Selection Sort

- Similar to insertion sort
- Performs worse than insertion sort in general
- Noted for its simplicity and performance advantages when compared to complicated algorithms
- The algorithm works as follows:
  - Find the maximum value in the list
  - Swap it with the value in the last position
  - Repeat the steps above for remainder of the list (ending at the second to last position)

Sorting Preview: Selection Sort

11	3	27	5	16
11	3	16	5	27
11	3	5	16	27
5	3	11	16	27
3	5	11	16	27

- Time Complexity:
  - $O(n^2)$
- Space Complexity:
  - $O(n)$