CSCI 136
Data Structures & Advanced Programming

Lecture 9

Fall 2018

Instructors: Bills

Administrative Details

- Remember: First Problem Set is online
- Due at beginning of class on Friday
- Lab 3 Today!
  - You *may* work with a partner
  - Come to lab with a plan!
  - Answer questions before lab

Last Time

- Measuring Growth
  - Big-O
- Introduction to Recursion

Today

- More Recursion
- Mathematical Induction (Weak)
- Mathematical Induction (Strong)

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

Proving Properties of Recursive Algorithms

- Example: factorial
  - Prove that fact(n) performs exactly n multiplications
    - Certainly true when n = 0…
    - Also, if—for some n—fact(n) performs exactly n multiplications, then fact(n+1) clearly performs exactly those plus one more: n+1
    - But fact(0) performs 0 multiplications, so fact(1) performs, fact(2) performs 2, ….
  - Said differently
    - Base case: n = 0 returns 1, performing 0 multiplications
    - Assume that for some n, fact(n) performs n multiplications.
    - fact(n+1) performs one multiplication directly: (n*fact(n-1)). We know that fact(n) performed n multiplications., therefore fact(n+1) performed n+1 multiplications.

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. Whenever 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. Whenever 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

- 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

- 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

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

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≥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 + fib(n−1) + fib(n−2) > fib(n−1) + fib(n−2) = 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. Whenever 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 ≤ c(1+log 1)
- Assume that claim holds for some n ≥ 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. Whenever 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 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

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)