C5450: Algorithms & Analysis

Instructor: Zach Minster

11/20/11 9:05-10:55am

I. Topic 1: Sorting (Fundamental problem in CS)
   a. Sorting – deck of cards demo (two ways: can see/cannot see next card)
      i. COMPARISON sorting – we use comparison to compare the next two values
      ii. This is an algorithm – algorithm discussion:
         1. Algorithms are ways to systematically solve problems regardless of input
         2. Beautiful – languages are *just tools* (and nothing more) to express the algorithms that you create inside of your head
      iii. How efficiently is the card sort happening on n cards? Welcome to formal analysis – this is the fundamental problem
      iv. Sorting is a big problem – My TripAdvisor interview involved lots of questions about sorting (and time complexity in general) because scalability of sorts over a huge amount of data is critical
   b. Pessimism – The Big O and Time Complexity
      i. We always assume Murphy's Law ("Anything that *can* go wrong *will* go wrong)
      ii. **We only care about the terms that grow the fastest**
         1. Some algorithm that we know runs in $n^2 + 4n + 34$ time is $O(n^2)$ because n^2 is the only term that matters for large n
         2. $O(n^3)$ is still far worse than $O(n^2)$, and $O(2^n)$ is virtually unforgiveable
         3. **This is why we ignore constant time steps in our algorithm**
   c. Bubble Sort – Bulbasaur – Anecdote about one of my professors in high school who always used "bulbasaur" when he meant "bubble sort"
      i. What is an array? What can we store in one?
         1. The cards we saw are just like an array
         2. Arrays store lists of data of a single type
      ii. Most basic sort – BUBBLE SORT
         1. The answer is in the cards – think about the card demonstration
         2. Iteration vs. Recursion
            a. This is an iterative method – it works with two for loops
         3. Look at the Java demo code – we see *iteration* happening on two loops
         4. This gives us the time complexity informally – each pass through the deck takes n operations for the comparisons and we have to do this n times in the worst case, so we're going to need $n * n = n^2$ operations in the worst case…
         5. WORST CASE: What improvements can we make? But do they change the time complexity in the WORST CASE? Think pessimistically!
   d. Bogosort

i. What is it? – Demonstration: Throwing the deck across the room
ii. Entropy guarantees us a solution eventually – but when?
    1. O(∞) in the worst case

e. Insertion sort
    i. A portion of the list – a growing part of the list on the left beginning with the leftmost element – is sorted
        1. Move forward to the next unsorted data element and shuffle it into the part of the list on the left
        2. Continue until the left "sorted" side has all the elements in the list
    ii. It's an adaptive sort – so we get nicer performance, why?
        1. If it's already sorted, we only pass through once
        2. We only exchange items when they need to be exchanged, and we can stop when we're sorted completely
    iii. So we've improved it – did our time complexity change?
        1. **No.** The time complexity did not change. **Think about pessimism.** What's the worst case? If we're looking for ascending, it's when the list is already sorted in descending order, and vice versa. **In this case we'll need n^2 operations in the worst case again!**

f. Ugh, does anything work in better than $O(n^2)$?
    i. Divide and conquer, splitting the list!
    ii. This is called a MERGESORT
    iii. Mergesort runs in O(n*log(n)), but why?
        1. Does it make sense conceptually?
        2. Yes, it does – we split the list by powers of two, so we only do log(n) merge operations
        3. Problem: very difficult to tackle analysis formally, leading to…

II. Topic 2 – Recursion, Induction, Proofs!

a. Recursion Function Examples – Some classics
    i. Factorial: fact(n) = n*fact(n-1)
    ii. Sum of all integers 0-n (also called the summorial): sum(n) = n + sum(n-1)
    iii. Note: look at how similar factorial and summorial are!
    iv. Fibonacci Sequence
        1. Each element is the sum of the two elements found before it
        2. Formally: fib(n) = fib(n-1) + fib(n-2)
        3. Looks like: 1, 1, 2, 3, 5, 8, 13…

b. Closed Forms
    i. Start listing out terms, can you see a pattern?
        1. Sum of all integers… we can see that it eventually looks like: (n(n+1))/2
    ii. There's no really amazing method to come up with these – there's an online bank of integer sequences by AT&T (AT&T OEIS, found here: http://oeis.org/), etc etc…this isn't the focus of our course, so we won't dwell on it

iii. Fibonacci closed form: (5 + sqrt(5))*pow(1+sqrt(5)/2,n)/10 + (5-sqrt(5))*pow(1-sqrt(5)/2,n)/10
    1. OUCH – closed forms aren't always the easiest to find!

c. Proof by Induction!
    i. Show the example for the sum of all integers
    ii. We can prove lots of things with this, but we don't have time – great way to prove your closed form!
    iii. Steps:
        1. Make a predicate: here I let P(n) mean that the sum of all integers from 0 to n is (n(n+1))/2
        2. Prove **base case** – just like recursion – P(0) works, right?
        3. Now prove that P(k) -> P(k+1) – suppose that P(k) is true, then use that assumption to prove that P(k+1) *must be true*
        4. Remember the N64 games – it's like the domino effect – you prove it for the first one, then you prove that each one necessarily follows from the one before, and you have it proven over the whole set! Simple as that!

d. Back to Mergesort – pretty ridiculous proof (I'll spare the odious details)!
    i. The recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists)
    ii. I will give closed form: In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than $(n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1)$, which is between $(n \lg n - n + 1)$ and $(n \lg n + n + O(\lg n))$

e. Other Divide and Conquer algorithms
    i. Finding the square root through binary search
    ii. Generating all the subsets of a set – also called the **power set** of a set (yay set theory!)

III. Course conclusion – Jerry's final thought (see my email for similar sentiments)

a. Congratulations! This is college level material, and I'm really impressed with how far we got in these two hours.