Self Assessment

Matt Forbes

December 8, 2010

Introduction

Thinking and working through problems this quarter has made been very satisfying. Challenge has been very prominent in just about every aspect of this course. Not only have the actual problems been difficult, but keeping my time managed while not losing quality has been as well. Trying to wrap my head around problems in this class was at first intimidating, but throughout the quarter I've developed a toolset and way of thinking that makes them much more managable.

Class Participation

After a rather tough bi-weekly assignment, I was chosen to present and explain the idea behind one of my solutions to the class. This is easier said than done. Actually getting up in front of a full class and trying to convey a hard-to-visualize thought is a bit nerve-racking. Luckily, I noticed that 'excellent' was written (and underlined twice!) next to my solution, so I was nearly positive I was going up. Knowing in advance gave me a bit of time to mentally prepare, but I don't have much experience in front of a class.

My solution was 'excellent' for the reason that I made sure to not use any assumptions without showing they were true. The problem was a divide and conqueror algorithm, but to prove that you could split the problem in half relied on an extra piece of information that I provided.

Context (from homework 2):

> If we take the median card from both hands, and call card **a** the smaller of the two, and **b** the greater. There are <u>at most</u> $n - \frac{n}{2}$ cards greater than **a** in its own hand. In **b**'s hand, there are at most $n - \frac{n}{2} - 1$ cards greater than **a**, because $\mathbf{a} > \mathbf{b}$. In total, there are <u>at most</u> $n - \frac{n}{2} + n - \frac{n}{2} - 1 = 2n - n - 1 = n - 1$ cards greater than **a**. Therefore, **a** must be greater than or equal to the absolute median of the 2n cards.
>
> Likewise for **b**.

Asymptotic Analysis

'Big-O' notation wasn't brand new to me coming in to this class, but my understanding was at a very simple level. It is generally easy to analyze basic algorithms like for loops, but I came to appreciate Big-O when analyzing more complicated forms like recursion.

Also from the card problem in homework 2, I used Big-O analysis to determine the runtime of my recursive algorithm. To bound the running time of my algorithm, I used a recurrence, and then bounded that recurrence from above with a 'Big-O value.' Demonstrating asympotic analysis with a recurrence:

> Let T(n) be the running time of this algorithm, where n is the total number of cards we are searching through.
>
> $$T(n) = \left\{ \begin{array}{ll} T(\frac{n}{2}) + O(1), & \text{for n} > 2 \\ \text{O(1)}, & \text{for n} \leq 2 \end{array} \right\}$$
>
> An upper bound guess for T is $T(n) \leq c \log_2(n) + O(1)$
>
> $$\begin{aligned} T(n) &\leq c \log_2(\frac{n}{2}) + O(1) \\ &\leq c \log_2(n) - \log_2(2) + O(1) \\ &= c \log_2(n) + O(1) \end{aligned}$$
>
> So, $T(n)$ is $O(\log_2(n))$.

Finally an example of setting up a summation to find an upper bound of a running time. From homework 3: problem 3, in the first part I was required to find the running time of searching some kind of array of sorted arrays. So from my writeup:

> Each array A_i has 2^i elements, so a binary search on that array would have a running time of $\log_2(2^i) = i$. The total running time t of the search procedure would be:
>
> $$\begin{aligned} \text{let k} &= \lceil logn \rceil \\ t &= \sum_{i=0}^{k} \log_2 2^i \\ &= \sum_{i=0}^{k} i \\ &= \frac{k(k+1)}{2} \\ &= \frac{1}{2}\left(\log^2 n + \log n\right) \\ &= O(\log^2 n) \end{aligned}$$

Loop Invariants

When I was first introduced to loop invariants, they almost seemed overly verbose and overkill. Their merit didn't seem unwarranted, but they didn't look very fun to me at all. Throughout the quarter we used a few of these to prove some less intuitive algorithms were correct. There was no way to just look at these algorithms we proved and be able to infer their correctness, so having a tool that could strongly state this was appealing.

My only attempt at writing a loop invariant didn't go as well as I might have hoped. My problem was ambiguity, it was not immediately obvious why my invariant worked (if it did at all.) The idea was right, but the execution wasn't.

Even though my own invariant wasn't perfect, I was able to correctly prove a pre-written one. An example of this would be on homework two problem one.

Probability and Indicator Random Variables

Having not really worked with probability before, the power of indicator random variables was surprising. Even trying to come up with a rough estimate for the running time of probabilistic algorithms is tough, but using IRAs make things so much more simple. It reminds me of 'divide and conquer' in the sense that it breaks down the complexity of the problem in to manageable parts and then aggregates them.

Analysis of Data Structures

We spent a lot of time in class analyzing the heap data structure. I knew about them, and learned how they worked at least twice but it never stuck with me. Without an analysis of why they are good at what they do, it's not hard to see why. Heaps have one goal which is to be able to return the min/max element in the structure extremely fast, and that's about it (besides inserting). Before the analysis they seemed pointless because in a balanced search tree, the same element can be accessed in $O(\log n)$ time. Now that we've discussed heaps, I notice that they can be very useful and extended to provide more functionality and stay very fast.

Amortized analysis is very useful tool, it picks up the slack when the usual worst case analysis falls short. For example, when we looked at hash tables. The tables doubled in capacity when they became half full. Using worst case analysis, you would say that an insert was $O(n)$ because in the worst case it has to rebuild the whole table. Obviously this is wrong, and amortized analysis came to the rescue where we can show that the average running time is still $O(1)$ for insert even though there are a few bad cases sprinkled in.

In the final homework, I used amortized analysis to find the average running time of operations on a special queue. Rather than using an internal list or array to store elements, it only has two stacks to work with. I was able to come up with an algorithm that is cheap for nearly all of its operations, but has an expensive one very once in a while. From there

I used amortized analysis to show that in general it has a quick running time. Here's an excerpt:

> In practice, we could not possibly have to do this mass popping and pushing to reorient the structure every dequeue. This means that we will have a much faster amortized analysis of the running time. If we follow the lifetime of one element in the structure, there are only about 4 operations associated with it. We initially push it on to the 'pushstack' and then at some later time we will transfer it to the 'popstack' and finally pop it one more time when it is removed. So there can never be more than 4 operations per element lifecycle. Using amortized analysis, we can see that n insertions could never be worse than about $4n$ stack operations. Therefore, the average cost per enqueue/dequeue operation is $\frac{4n}{n} = 4$, which is $O(1)$.

Augmented Data Structures

Textbook data structures work well for basic applications, but in this class we found cases where they didn't quite cut it. For example, we wanted to store intervals in some sort of binary tree format and easily look for overlaps. An augmented binary search tree lent itself well to this problem. Every node held two keys, one for the start time and one for the end. Now we had all the functionality of a binary search tree, but when we implemented the function for checking the tree for overlapping intervals, it was a piece of cake.

A more useful example of an augmented structure we used was an order statistic tree. This tree is similar to the interval tree in that it stores and extra (augmented) field along with each node. In this case it was the number of child nodes in its left subtree. With this augmentation, we were able to add extra functionality to the structure without breaking the running time of its core functions (insert, delete, search, etc.). Given any node in the tree, we could figure out its rank in $O(\log n)$ time, and similarly we could find the node corresponding to any rank (if it exists) in the same amount of time. I used this structure in assignment four.

Sorting Algorithms

Two of the sorting algorithms we discussed are merge sort and quick sort. Merge sort is a naturally recursive algorithm which has a running time of $O(n \log n)$ but ends up creating and destroying a lot of auxillary data. Analyzing merge sort would be hard if we hadn't used recurrences, which describe the problem in terms of smaller versions of the problem. For example, the recurrence for the running time of merge sort might look like this:

$$M(n) = \left\{ \begin{array}{ll} M(\frac{n}{2}) + M(\frac{n}{2}) + O(1) & n > 1 \\ O(1) & n = 1 \end{array} \right\}$$

On the other hand, quick sort can be done in place. This has several advantages that didn't necessarily show up in our analysis of it's running time. Because it isn't constantly creating and destroying memory, it runs significantly faster than merge sort, even though its worst case running time is slower ($O(n^2) vs O(n \log n)$). I suppose that is one of the downfalls of using the generic Big-O notation that doesn't differentiate the cost of different basic oeprations.