CS2040S: Data Structures and Algorithms

Tutorial 2

Problem 1. Queues and Stacks

Queues have a lot of practical uses. This exercise will go through some of these uses:

(a) Last week, we learnt about the basic operations of stacks and queues. How would you implement a stack and queue in Java?

(b) A set of parentheses is said to be balanced as long as every opening parentheses "(" is closed by a closing parentheses ")". So for example the strings "()()" and "(())" are balanced but the string ")(())(" and "((" are not balanced. Using a stack, determine whether a string of parentheses are balanced.

(c) Sort an array using two queues.

(d) (Challenge) Implement a queue that allows you to get the minimum item as efficiently as possible.

Problem 2. Moar Pivots!

Quicksort is pretty fast. But that was with one pivot. Can we improve it by using two pivots? What about k pivots? What would the asymptotic running time be? (That is, the algorithm is to choose the pivots at random—or perhaps, imagine you have a magic black box that gives you perfect pivots—then sort the pivots, partition the data among the pivots, and recurse on each part. You may assume whichever gives you a better performance)

Problem 3. It's Not Just About Time

What if your goal is to minimize the number of times data is written, rather than the number of comparisons? Assume you want your algorithm to be in-place. What is a good algorithm in that case? Assume for now you do not care about comparisons at all. One case where this is important is if you have very large data to sort: comparing is relatively cheap (as you only have to look at a small prefix of the data to decide the order, in most cases), but moving is expensive (because you have to re-write a large file.)

Problem 4. But Wait There's More...

Continuing on from Problem 3, now your goal is to keep $O(n)$ writes, but with only $O(n \log n)$ cost for reads. (This turns out to be important for non-volatile NVRAM memory where writing takes longer than reading, but both matter.) For now, do not worry about the algorithm being in-place, but be sure to count every single write operation. (E.g., if you create an auxiliary array and write an integer to that array, it counts.)

Problem 5. Child Jumble

Your aunt and uncle recently asked you to help out with your cousin's birthday party. Alas, your cousin is three years old, and so that means spending several hours with twenty rambunctious three year olds, as they race back and forth, covering the floors with paint and hitting each other with plastic beach balls. Finally it is over. You are now left with twenty toddlers that each need to find their shoes. And you have a pile of shoes that all look about the same. They are not helpful. (Between exhaustion, too much sugar, and being hit on the head too many times, they are only semiconscious.)

Luckily, their feet (and shoes) are all slightly different size. Unfortunately, they are all very similar, and it is very hard to compare two pairs of shoes or two pairs of feet to decide which is bigger. (Have you ever tried asking a grumpy and tired toddler to line up their feet carefully with another toddler to determine which has bigger feet?) So you cannot compare shoes to shoes or feet to feet.

The only thing you can do is you can have a toddler try on a pair of shoes. When you do this, you can figure out whether the shoes fit, or if they are too big, or if they are too small. That is the only operation you can do.

Come up with an efficient algorithm to match each child to their shoes. Give the time complexity of your algorithm in terms of the number of children.

1) Use one random toddler as a "pivot" and apply quicksort routine once. (n operation)
2) Now line up on the Left are shoes that are smaller, right are shoes that are bigger. The one shoe that fits is now sorted and thus removed from the consideration.
3) Use that same pair of shoes to determine whether each toddler is smaller or bigger.(n operation)
4) Now repeat on the smaller group of toddlers steps 1-3 (which will take log2 n divisions)
Overall it is a O(n log n) operation