CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 4

For: February 1–February 5

Problem 1. Sorting Review

(a) How would you implement insertion sort recursively? Analyse the time complexity by formulating a recurrence relation.

(b) Suppose that the pivot choice is the median of the first, middle and last keys, can you find a bad input for QuickSort?

(c) Are any of the partitioning algorithms we have seen for QuickSort stable? Can you design a stable partitioning algorithm? Would it be efficient?

(d) Consider a QuickSort implementation that uses the 3-way partitioning scheme (i.e. elements equal to the pivot are partitioned into their own segment).

i) If an input array of size n contains all identical keys, what is the asymptotic bound for QuickSort?
For example, you are sorting the array $[3, 3, 3, 3, 3, 3]$

ii) If an input array of size n contains $k < n$ distinct keys, what is the asymptotic bound for QuickSort?
For example, with $n = 6, k = 3$, sort the array $[a, b, a, c, b, c]$

(e) Consider an array of pairs (a,b). Your goal is to sort them by ascending a first, and then by ascending b. For example, (1,3), (1,4), (2,1).... You are given 2 sorting functions. One that does a mergesort and one that does a quicksort.You can use each one at most once. How would you sort the pairs? Assume you can only sort one field at a time.

Problem 2. Queues and Stacks Review

Recall the Stack and Queue Abstract Data Types (ADTs) that we have seen in CS1101S. Just a quick recap, a Stack is a "LIFO" (Last In First Out) collection of elements that supports the following operations:

- **push**: Adds an element to the stack
- **pop**: Removes the **last** element that was added to the stack
- **peek**: Returns the last element added to the stack (without removing).

And a Queue is a "FIFO" (First In First Out) collection of elements that supports these operations:

- **enqueue**: Adds an element to the queue
- **dequeue**: Removes the **first** element that was added to the queue

- **peek**: Returns the next item to be dequeued.

(a) How would you implement a stack and queue with a fixed-size array in Java? (Assume that the number of items in the collections never exceed the array's capacity.)

(b) A Deque (double-ended queue) is an extension of a queue, which allows enqueueing and dequeueing from both ends of the queue. So the operations it would suppport are *enqueue_front*, *dequeue_front*, *enqueue_back*, *dequeue_back*. How can we implement a Deque with a fixed-size array in Java? (Again, assume that the number of items in the collections never exceed the array's capacity.)

(c) What sorts of error handling would we need, and how can we best handle these situations?

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

(e) (Optional) Sort a queue using another queue with $O(1)$ additional space.

Problem 3. 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 that 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 4. 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. 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. The toddlers are not helpful. (Between exhaustion, too much sugar, and being hit on the head too many times, they are only semi-conscious.)

Luckily, their feet (and shoes) are all of slightly different sizes. 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 who has bigger feet?) As such, you cannot compare shoes to shoes or feet to feet.

The only thing you can do is to 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 too small. That is the only operation you can perform.

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.