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.

Solution: Recursively sort the array from index 0 to (n - 1). Then, insert the last element into the sorted subarray. The recurrence relation will be $T(n) = T(n-1) + O(n)$. Solving it results in the time complexity $O(n^2)$.

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

Solution: As long as we have a fixed pivot choice, the time complexity would remain at $O(n^2)$ as it is always possible to find a bad input for the algorithm.

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

Solution: All partitioning algorithms are not stable. We can make them so by associating the original indices of each key with the key - a simple way would be to create an auxilliary array of indices which swaps will be performed on too. When comparing elements, this would be used to disambiguate elements with equal keys, creating a "total ordering" between every key.
Note that means that the partitioning algorithm is no longer in-place.

(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 $[3, 3, 3, 3, 3, 3]$

Solution: It should just take $O(n)$ time (even in the worse case), as after the first partitioning pass (which takes $O(n)$), the "unsorted" segments would be empty.

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 $[a, b, a, c, b, c]$

Solution: We can think of each branch in the recursion tree of QuickSort as a result of 1 pivot. As there are only k distinct keys, up to k pivots would be chosen in the whole QuickSort run, and so there are $O(k)$ branches in our recursion tree. This bounds the height of our recursion tree (in the worst case) to be $O(k)$. As we have no information on how many of each key we have, we can only assume that at every level of the tree, $O(n)$ time would be used for partitioning. So, putting them together the asymptotic bound should be $O(nk)$.
If our pivot selection is guaranteed to be balanced, it should be $O(n \log k)$

(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.

Solution: You should use the quicksort to sort the pairs by their b, and then use mergesort to sort the pairs by their a. This is because, mergesort is stable and would not affect the ordering of the b which is already sorted if their a is the same.

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.

Solution: The main point of this question is to go over the Stack and Queue APIs as this might be the last time we get to focus on them. It is important to fully understand these two structures since they will be used a lot in future algorithms (e.g. DFS and BFS). Tutors are free to practice with other questions apart from these (e.g. Expression evaluation).

(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.)

Solution: **Stack**: Maintain *top_index* that represents the index of the top of the stack. This should start with 0 assuming that we start with an empty array. To perform a *push*, place the new element at the current top index and increment it. To perform a *pop* or *peek*, return *top_index* $-$ 1, (and decrement it in a *pop* operation.). All of these operations are achievable in $O(1)$ time.
Queue: We need to maintain two indices, *head* and *tail* to represent the head and tail of the queues respectively (again, both starting at 0). To perform *enqueue*, place the new element at *tail* and increment *tail*. To perform *dequeue* or *peek*, return the element at *head* (and increment *head* in a *dequeue* operation). Again, all the operations would be in $O(1)$.
We might find that after performing $l-1$ enqueues and $k < l$ dequeues (where l is the array's capacity), we would be enqueuing at index l which is out of the array's index bounds. At the same time, we find that array indices 0 to $k-1$ are all unused. The idea here is to "wrap around" and use 0 as the next index, so we can reuse the unused spaces. A simple way to achieve this is to replace increments `x + 1` with `(x + 1) % array_length`.

(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.)

Solution: The idea is the same with the queue implementation, except any *_front* operations would manipulate the *head* index, while *_back* operations would manipulate the *tail* index.

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

Solution: For *enqueue* and *push* operations, we mostly need to be mindful of our underlying array's capacity. If the current collection size is equal to the array's capacity, we can report that the operation is invalid (by returning `false`, for example). Alternatively, we can even think of a strategy to move to an array with a bigger capacity (this might involve an expensive operation to copy everything in this array, but as we will learn later on in CS2040S, it may not be that costly after all). Detecting this error is relatively simple for our stack implementation (we just need to check if *top_index* is equal to array's length), but might be a bit tricker for our "circular" array queue.
For *dequeue* or *pop* operations, the main error to check for is that we are not performing on an empty collection - if we find that this is so, we would need to throw an exception again.

(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.

Solution: The idea behind this is to simply push opening parentheses onto a stack, then pop them out whenever you encounter a closing parenthesis. The string is unbalanced if there is an attempt at popping an empty stack, or if the stack is non-empty after reading the last character. As an extension, think about how you would tackle the case where there are multiple different type of parentheses such as {} and [].

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

Solution: The näive way to do this is by simply cycling through the queue, picking the minimum element each time, and appending it to the second queue. However, we can of course do better. The idea is to do it like merge sort! We will start by sorting by pairs, then by 4 elements, so on and so forth. For example, consider the following queue, $Q1$:

Head 19 8 16 5 10 15 18 9 7 Tail

In the first iteration, we group the elements into groups of $2^1 = 2$ and sort within each group. To sort the first pair, we dequeue the first half of the pair (in this case, 19) from $Q1$ and enqueue it in the second queue, $Q2$. Then, we compare the the heads of both queues. The smaller element (in this case, 8) gets dequeued and enqueued in $Q1$. Then, the other element (in this case, 19) gets dequeued and enqueued in $Q1$. We do this for every pair of elements until 7 is at the head of $Q1$. At this point in time, the elements in $Q1$ are:

Head 7 8 19 5 16 10 15 9 18 Tail

Since there are an odd number of elements in $Q1$, 7 does not have a paired element to compare against. As such, we just dequeue and enqueue 7 back into $Q1$. We thus end the first iteration with the following state of $Q1$:

Head 8 19 5 16 10 15 9 18 7 Tail

In the second iteration, we group the elements into groups of $2^2 = 4$ and sort within each group. To sort the first group of 4, we dequeue the first half of the group (in this case, 8 and 19) from $Q1$ and enqueue it in $Q2$. Then we compare the heads of both queues, dequeuing the smaller element and enqueuing into $Q1$ **until both queues have had** $4/2 = 2$ **dequeue operations**. By now, the state of $Q1$ is:

Head 10 15 9 18 7 5 8 16 19 Tail

We continue this process and end the second iteration off with $Q1$ being in the following state:

Head 5 8 16 19 9 10 15 18 7 Tail

After two more iterations, $Q1$ will be sorted. This algorithm runs in $O(n \log n)$ time because there are $O(\log n)$ iterations, each with $O(n)$ dequeue and enqueue operations.

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)

Solution: Notice that partitioning now takes:

- $O(k \log k)$ time to sort the pivots.
- $O(n \log k)$ time to place each item in the right bucket (e.g., via binary search among the pivots).

So the resulting recurrence is: $T(n) = kT(n/k) + O(n \log k)$, as long as $n > k$. The solution to this recurrence is $O(n \log_k n \log k) = O(n \log n)$, i.e., no improvement at all! Worse, doing the partition step in place becomes much more complicated, so the real costs become higher.
Except — and here's the amazing thing — we have discovered that 2 and 3 pivot QuickSort really is faster than regular QuickSort! Try running the experiment and see if it's true for you! Currently, the Java standard library implementation of QuickSort is a 2-pivot version!
Let's quickly consider how a partitioning algorithm can look with the 2-pivot Quicksort, which partitions a segment into 3 segments: elements `< pivot1`, `>= pivot1 && <= pivot2` & `> pivot2` (Assuming w.l.o.g that `pivot1 < pivot2`).
This should look familiar to another 3-way partitioning scheme that we have seen before - consider the partitioning scheme where elements equal to a pivot are also assigned a segment. We simply reuse this algorithm (by replacing the conditions) to create an in-place partitioning algorithm for 2-pivot quicksort. You can see this exact algorithm in action in Java's QuickSort implementation.
Students are encouraged to try running a few experiments on their own with regards to this.

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.

Solution: This is a classic QuickSort problem, often presented in terms of nuts-and-bolts (instead of kids). The basic solution is to choose a random pair of shoes (e.g., the red Reeboks), and use it to partition the kids into "bigger" and "smaller" groups. Along the way, you find one kid ("Alex") for whom the red Reeboks fit. Now, use Alex to partition the shoes into two piles: those that are too big for Alex, and those that are too small. Match the big shoes to the kids with big feet, the small shoes to the kids with small feet, and recurse on the two piles. If you choose the "pivot" shoes at random, you will get exactly the QuickSort recurrence.