CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 5

For: February 6–February 10

1 Check in and PS3

Discuss questions, if you have any, with the tutor and the rest of the class, about the material and content so far.

Solution: Your goal here is to find out how the students are doing. One idea: ask each of them to send you (perhaps anonymously) ONE question before tutorial about something they are confused by. For questions that make sense to answer as a group, ask the students to explain the answers to each other. (For questions that are not suitable for the group, offer to answer them separately.)

2 Problems

Problem 1. QuickSort Review

(a) 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.
In the case of the scenario given above, the key is to always select the border values (i.e. the first or last element) as pivots, so that the recursion tree is imbalanced and only 2 elements can be removed per partition. For example (the underlined section indicates the subarray that is currently being recursed on, while the bolded keys are the first, middle, and last keys):

1st Partitioning : $[\underline{\mathbf{8}, 3, 2, 1, \mathbf{5}, 4, 6, 7, \mathbf{9}}]$ (8 will be selected as the pivot)
2nd Partitioning : $[\underline{\mathbf{7}, 3, 2, \mathbf{1}, 5, 4, \mathbf{6}}, 8, 9]$ (6 will be selected as the pivot)
3rd Partitioning : $[\underline{\mathbf{4}, 3, \mathbf{2}, 1, \mathbf{5}}, 6, 7, 8, 9]$ (4 will be selected as the pivot)
4th Partitioning : $[\underline{\mathbf{1}, \mathbf{3}, \mathbf{2}}, 4, 5, 6, 7, 8, 9]$ (2 will be selected as the pivot)

(b) 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. However, we can make them stable by associating the original indices of each key with the key - a simple way would be to create an auxiliary array of indices which swaps will be performed on too.

$$\text{Original Array} : [1, \mathbf{2}, 5, 3, 5, 3, 8, 7, \mathbf{2}]$$
$$\text{Auxiliary Array} : [0, \mathbf{1}, 2, 3, 4, 5, 6, 7, \mathbf{8}]$$

When comparing elements, the auxiliary array would be used to disambiguate elements with equal keys, creating a "total ordering" between every key. For example, when comparing the two 2s in the original array, the sorting algorithm will take a look at the auxiliary array to determine which value came first in the original array (1 or 8).
Note that by doing so, the partitioning algorithm is no longer in-place since an auxiliary array is needed to store the original indices.

(c) 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?

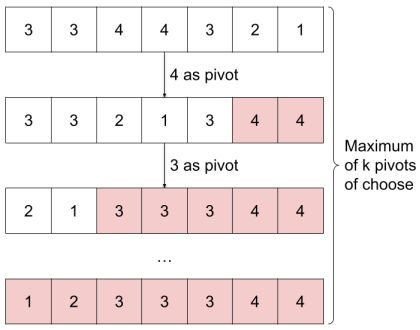
For example, you are sorting the array $[3, 3, 3, 3, 3, 3]$

Solution: It should always take $O(n)$ time, 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 level in the recursion tree of QuickSort as a result of partitioning using only 1 distinct pivot. As there are only k distinct keys in the array, up to k pivots would be chosen in the whole QuickSort run, and so 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, the asymptotic bound should be $O(n \log k)$.

Problem 2. (A few puzzles involving duplicates or array processing)

For each problem, try to come up with the most efficient algorithm and provide the time complexity for your solution.

(a) Given an array A, decide if there are any duplicated elements in the array.

Solution: First, we sort the array. This takes $O(n \log n)$ time. Then, we traverse the array from index 0 to $n-2$, checking whether the value at index i is the same as the value at index $i+1$. This takes $O(n)$ time. The overall runtime would thus be $O(n \log n)$.

(b) Given an array A, output another array B with all the duplicates removed. Note the order of the elements in B does not need to follow the same order in A. That means if array A is $\{3, 2, 1, 3, 2, 1\}$, then your algorithm can output $\{1, 2, 3\}$.

Solution: First, we sort the array in ascending order, taking $O(n \log n)$ time. Then, we traverse the array while using a variable to keep track of the largest element k encountered so far. We remove the element at index i if it is identical to k, or update the value of k if the element at index i is not a duplicate of previous elements. The overall runtime would be $O(n \log n)$.

(c) Given arrays A and B, output a new array C containing all the distinct items in both A and B. You are given that array A and array B already have their duplicates removed.

Solution: We can employ the same idea as in merge sort. Sort both arrays in ascending order, then use the merge step from merge sort to output array C. If the element has already been added to array C, we discard the element, and advance our pointer in array A or array B. The runtime would be $O(n \log n)$.

(d) Given array A and a target value, output two elements x and y in A where $(x+y)$ equals the target value.

Solution: Sort the array. Use two pointers to mark the beginning and end of the array respectively. If the target is less than the sum of those two elements, shift the left pointer to increase the value of $(x+y)$ and if the target is greater than the required value, shift the right pointer to decrease the value. The runtime would be $O(n \log n)$.

Problem 3. 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, which results in a runtime of $O(n \log n)$ where n is the number of children.

Problem 4. More Pivots!

QuickSort is pretty fast. But that was with one pivot. In fact, QuickSort can also be implemented with two or more pivots! In this question, we will investigate the asymptotic running time of QuickSort when there are k pivots.

(a) Suppose that you have a magic black box function which chooses k perfect pivots that separate the elements evenly (e.g. it picks the quartile elements when there are 3 pivots). How would a partitioning algorithm work using these pivots?

Solution: First, sort the pivots. Then for each element, use binary search to place it in the correct partition. 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`, and `> pivot2` (Assuming w.l.o.g that `pivot1 < pivot2`).

[**57** 8 42 75 29 77 38 **24**]

Here we choose 57 and 24 as our 2 pivots. We first sort the pivots, so that the rest of the elements can take advantage of binary search and be placed in their correct segments. Let's take a look at how the array will look like after the first partitioning.

[8 **24** 42 38 29 **57** 75 77]

After partitioning, pivots 24 and 57 are guaranteed to be in the correct index. Next we can recurse in the left segment (8), middle segment (42 38 29), and right segment (75 77). 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.

(b) What is the asymptotic running time of your partitioning algorithm? Give your answer in terms of the number of elements, n and the number of pivots, k.

Solution: Notice that partitioning now takes:

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

We have $O(k \log k) + O(n \log k) = O(n \log k)$ as long as $n \geq k$.

(c) We can implement QuickSort using the partitioning algorithm you devised by recursing on each partition. Formulate a recurrence relation that represents the asymptotic running time of quicksort with k pivots. Give your answer in terms of the number of elements, n and the number of pivots, k.

Solution: Since QuickSort recurses on k partitions with n/k elements each, we have $T(n) = kT(n/k) + O(n \log k)$.

(d) Solve the recurrence relation to obtain the asymptotic running time of your QuickSort with k pivots. Would using more pivots result in an improvement in the asymptotic running time?

Solution: The solution to this recurrence is $O(n \log k \log_k n) = O(n \log n)$, 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! This is an example where performance in theory does not translate into real life. It happens because 2-pivot quicksort takes benefits of modern computer architecture and has reduced cache misses. *Students are encouraged to try running a few experiments on their own with regards to this.*

Problem 5. Integer Sort

(a) Given an array consisting of only 0's and 1's, what is the most efficient way to sort it? (Hint: Consider modifying a sorting algorithm that you have already learnt to achieve a running time of $O(n)$ regardless of the order of the elements in the input array.)

Can you do this in-place? If it is in-place, is it also stable? (You should think of the array as containing key/value pairs, where the keys are 0's and 1's, but the values are arbitrary.)

Solution: We can do this using QuickSort partitioning with two pointers, one at each side. The 0-pointer will advance to the right up until it finds the first 1; the 1-pointer will advance to the left up until it finds the first 0. Then, swap the 1 found on the left with the 0 found on the right. Repeat this until the all the elements in the array have been visited (the pointers meet each other), and the running time is $O(n)$.
Note that in-place partitioning (as done by QuickSort) is unstable. You can make it stable by copying to another array, but that will incur additional space complexity.

(b) Consider an array consisting of integers between 0 and M, where M is a small integer (For example, imagine an array containing key/value pairs, with all keys in the range $\{0, 1, 2, 3, 4\}$). What is the most efficient way to sort it? This time, you do not have to do it in-place; you can use extra space to record information about the input array and you can use an additional array to store the output.

Solution: We can do this using Counting Sort.

- First, go through the array once counting how many of each element you have.
- Then, go through the array and compute where the first element for each value should go in the output array. For example, to find out where the first '3' in the array goes, sum up how many 0's, 1's, and 2's there are in the input array: if there are 5 0's, 3 1's, and 4 2's, then the very first 3 is going to go in slot $A[12]$ of the output array A (counting from zero). This can easily be computed from the array computed in the previous step.
- Treat this new array as a set of M pointers that point to the beginning of each block for each value. Now iterate through the input array and copy each item into the proper place indicated by the pointer. Then advance the pointer.

This takes M space to keep track of how many of each item there are, and M space to keep track of the pointers to each region of the array.

(c) Consider the following sorting algorithm for sorting integers represent in binary (each specified with the same number of bits):

First, use the in-place algorithm from part (a) to sort by the first (high-order) bit. That is, use the high-order bit of each integer as the key to sort by. Once this is done, you have divided the array into two parts: the first part contains all the integers that begin with 0 and the second part contains all the integers that begin with 1. That is, all the elements of the (binary) form '0xxxxxxx' will come before all the elements of the (binary) form '1xxxxxxx'.

Now, sort the two parts using the same algorithm, but using the second bit instead of the first. And then, sort each of those parts using the 3rd bit, etc.

Assuming that each integer is 64 bits, what is the running time of this algorithm? When do you think this sorting algorithm would be faster than QuickSort? If you want to, write some code and test it out.

Solution: It will take about 64n steps. Each level takes $O(n)$ time to be sorted, and we repeat this for each bit in the 64 bit integers, making the recursion at most 64 levels deep. It is in-place, and just involves scanning the array 64 times, so it is fairly efficient in a lot of way. Unfortunately, since QuickSort runs in approximately $\Theta(nlogn)$ time, this will only likely beat QuickSort when $n > 2^{64}$, which is quite large!

(d) Can you improve on this by using the algorithm from part (b) instead to do the partial sorting? What are the trade-offs involved?

Solution: For example, you might divide each integer up into 8 chunks of 8 bits each. You can use part (b) algorithm to do the sorting where the values range from 0 to 255. This will still take $O(n)$ time for each partial sort. Now the "recursion" only goes 8 levels deep. This now has a chance of being faster than QuickSort, though it might still take a pretty large input at $n > 2^8$ to do so.
Since the algorithm is not in-place, the trade-off we make is space. It will take 256 integers worth of space to do the sorting using the part (b) algorithm. (Also, the fact that it is not in-place may make it slower than you would expect.)

Problem 6. (If you have time)
What solutions did you find for Contest 1 (Treasure Island)?

Solution: First, review the binary search approach. This should give a solution of cost $O(k \log n)$. Each binary search of cost $\log n$ finds one more correct key.
Here's one approach for doing better:

- Choose a set of n/k keys that are still unexamined and put them in set S.
- Check if there is at least one correct key in S by using all the keys not in S to unlock the chest.
- If there is at least one correct key in S, then repeatedly use binary search to find the correct keys in S. Each correct key you find here will take $O(\log(n/k))$ time.
- Mark the keys in set S as examined, and repeat with the remaining keys.

Since each correct key you find takes $O(\log(n/k))$ time, you will spend $O(k \log(n/k))$ time doing binary searches for correct keys. How many times will you query a set S and discover no correct keys? Well, there are only k different sets with n/k keys, so this will take at most k queries. So the total number of queries is $k \log(n/k) + k$. This is optimal, since $\Omega(k \log(n/k))$ is the best you can do.

Solution: Here are some more notes on Contest 1. First, let's review the binary search solution to treasure island.
At any given time, we have a set K of keys already discovered, a set S of all the other keys, and a set C of keys we are curious about. Initially, K is empty, S is empty, and C is all the keys.
One invariant here is that C must contain at least one key at all time.
Repeatedly:

- Let T = half the keys in C.
- Take all the keys in T, all the keys in K, and all the keys in S. Try those!
- If it does not open, then there is at least one key in $C - T$. So delete the keys T from C and add the keys T to S.
- If it does open, then there are no keys in $C - T$. So add all the keys in $C - T$ to S, and set $C = T$.
- Eventually, there is only one key left in C. Add that to K, and then put all the keys in S back in C, and set $S = \emptyset$.

Notice that after $\log(n)$ tries, you will find at least one key. So after $k \log(n)$ tries, you will find all k keys. (In fact, it is just a little faster, because some of the keys are already in K toward the end.)
Now, let's try to do better. Instead, every time you begin finding a new key, do as follows:

- Choose n/k keys. Call that set R.
- Try all the keys NOT in R. If it fails, then we know there is at least one key in R. At this point, set $C = R$, S =all the other keys that are not in R or K, and run the binary search algorithm from above.
- If it succeeds, then all the keys are not in R. Delete the keys in R, and try again.

Notice that you choose sets R at most $2k$ times: either you delete n/k items, or you find a key, and each of those things can happen at most k times. Also, you run the binary search at most k times, and each time it costs at most $\log(n/k)$, since there are at most n/k items in C. So the reason we chose R to be n/k was to balance the two cases: if there is a key in R, we can find it in $\log(n/k)$ time; and if there is no key in R, then we can eliminate enough items (to ensure that case doesn't happen more than k times).

The natural question is whether you can make R smaller. If you make R too much smaller, then most of the time you won't find a key there. But it might look like you could do better by choosing R to be a little smaller (e.g., a factor of $\log(n/k)$ smaller). But it turns out that, asymptotically, that won't help. However, you can certainly adjust the size of R to fine-tune for constant factors! I think you will definitely do (constant-factor) better for R a little bit smaller. If I were competing in the contest, I'd probably try some smaller sets R.

You could also try partitioning the keys into k sets of size n/k, then spending k tries to identify which contain at least one key. About $1/e$ of the groups will have exactly one key, which you can then binary search. (This follows from a balls-in-bins strategy: if you throw k balls at random into k bins, then in expectation k bins have exactly one ball. Here we can ignore all the useless keys.) That works too! And there are a few other similar strategies. Asymptotically, $\Theta(k \log(n/k))$ is actually the best you can do. But in practice, there is definitely some optimisation to be done.