CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 5

For: February 10–February 14

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.

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 arrange the array in such a way that the second-smallest (or second-largest) element is always selected as pivot, 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):

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

(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 where swaps will be performed 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 2's 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?

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?

Solution: Because there are only k distinct keys in the array, up to k pivots would be chosen in the whole QuickSort run. As we have no other information on the array, we can only assume that $O(n)$ time would be used for each partitioning. So, putting them together the asymptotic bound should be $O(nk)$.
We can show that this is the lower bound of our algorithm as well. If the pivot choice is the first key, then a bad input for our algorithm would be $[1, 2, \ldots, k-1, k, k, k, \ldots]$. Similarly, we can construct bad inputs for other fixed pivot choices as well.

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 MergeSort. Sort both arrays in ascending order, then use the merge step from MergeSort 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 current sum of the elements pointed by the two pointers is less than the target value, move the left pointer forwards to increase the sum. If the current sum is greater than the target value, move the right pointer backwards to decrease the sum.
The algorithm terminates once the current sum is equal to the target, or when the left pointer meets the right pointer, in which case there is no solution. 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 who 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 that 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 would work with 2 pivots, which partitions a segment into 3 segments.

[**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 what 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: We can solve the recurrence as follows:

$$\begin{aligned}
T(n) &= c(n \log k) + kT\left(\frac{n}{k}\right) \\
&= c(n \log k) + k \cdot c(\frac{n}{k} \log k) + k^2 T\left(\frac{n}{k^2}\right) \\
&= c(n \log k) + k \cdot c(\frac{n}{k} \log k) + k^2 \cdot c(\frac{n}{k^2} \log k) + k^3 T\left(\frac{n}{k^3}\right) \\
&= c(n \log k) + k \cdot c(\frac{n}{k} \log k) + k^2 \cdot c(\frac{n}{k^2} \log k) + \ldots + k^{\log_k n} \cdot c(\frac{n}{k^{\log_k n}} \log k) \\
&= c(n \log k) + c(n \log k) + c(n \log k) + \ldots + c(n \log k) && (\log_k n \text{ times}) \\
&= c(n \log k) \cdot \log_k n \\
&= O(n \log k \log_k n)
\end{aligned}$$

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 advantage of modern computer architecture and has reduced cache misses. *You are encouraged to try running a few experiments on your own!*

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 on 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 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 create a stable algorithm by copying to another array, but that will incur additional space complexity.

(b) Consider an array A consisting of elements with keys being 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 set $\{0, 1, 2, 3, 4\}$). 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 CountingSort.

(1) Prepare a frequency array F with size $M + 1$ and initialize it with 0's.

(2) Go through the array once counting how many elements of each key you have, and update the frequency array accordingly. That is, for each i from 0 to $n - 1$, increment the value of $F[A[i]]$ by 1.

(3) Perform a prefix sum computation on F. For each i from 1 to M, set $F[i] = F[i-1]+F[i]$. Now, each $F[i]$ is equal to the last position where an element with key equal to i should be placed, plus 1.

(4) Create a new array B of size n and go through the original array backwards from $i = n-1$ to $i = 0$. For each i, decrement $F[A[i]]$ by 1. Then, place $A[i]$ at index $F[A[i]]$.

(5) The resultant array B is a stable sorting of the original array A.

This takes $O(M)$ space for array F, and $O(n)$ space for array B. The time complexity of this solution is $O(n + M)$.

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

(1) First, use the in-place algorithm from part (a) to sort by the most significant bit, or MSB. That is, use the MSB of each integer as the key to sort with.

(2) 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. In other words, all the elements of the (binary) form `0xxxxxxx` will come before all the elements of the (binary) form `1xxxxxxx`.

(3) Now, sort the two parts using the same algorithm, but using the 2nd bit instead of the 1st. Then, sort each of the resulting parts using the 3rd bit, and so on.

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: Each level takes $O(n)$ time to be sorted, and we repeat this for each bit of the 64-bit integers, making the recursion at most 64 levels deep. So the time complexity of our algorithm is $O(n)$.
Our algorithm is in-place, and just involves scanning the array 64 times, so it is fairly efficient in a lot of ways.
Unfortunately, even though QuickSort has a theoretically larger time complexity of $O(n \log n)$, its constant factor is much lower, meaning our algorithm will only likely beat QuickSort for extremely large values of n!

(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 the algorithm in part (b) 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 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.)