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?

The pivot selected (median of the first, middle and last keys) would need to be the smallest or largest value of the array.
For example, one array could be [8, 3, 2, 1, 5, 4, 6, 7, 9], where the first, middle and last keys are 8, 5, and 9 respectively. This would result in the median value of 8 being selected as the pivot. The subarray of [7, 3, 2, 1, 5, 4, 6, 8, 9] would then be recursed on with the first, middle and last keys being 7, 1 and 6. The median value of 6 would then be selected as the pivot and the subarray of [4, 3, 2, 1, 5, 6, 7, 8, 9] would be recursed on. This time, the keys are 4, 2 and 5, with the median value being 4. The resulting array would be [1, 3, 2, 4, 5, 6, 7, 8, 9], where 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?

No.
Instead of swapping elements in place, use additional arrays to separately collect the elements which are smaller than the pivot, equal to the pivot, and greater than the pivot. After selecting the pivot, when iterating through the array, place elements smaller than the pivot into the "less" array, elements equal to the pivot into the "equal" array, and elements larger than the pivot into the "greater" array. Recursively apply the partitioning algorithm to the "less" and "greater" array. Once the base case is reached, concatenate the "less", "equal" and "greater" arrays together to form the sorted array. The algorithm would still be efficient but would no longer be sorting the elements in place.

(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?

T(n) = O(n) as the pivot end index would just be incremented by 1 each time, until it reaches the n – 1 element.

ii) If an input array of size n contains $k < n$ distinct keys, what is the asymptotic bound for QuickSort?

Each level in the recursion tree of QuickSort can be thought of as partitioning using only 1 distinct key. Since there are k distinct keys, up to k pivots would be selected. This bounds the height of our recursion tree to be O(k). We assume that it takes O(n) time to partition the array at each level of the tree. Thus, the asymptotic bound for QuickSort would be O(nk). If we are able to guarantee the best pivot were to be selected when partitioning each time, the asymptotic bound would be O($n \log k$).

O($n \log k$) – think of it as we have k keys as pivots, and each time we recurse, we have $\left(\frac{k}{2}\right)$, $\left(\frac{k}{4}\right)$, $\left(\frac{k}{8}\right)$... distinct keys. Sooner or later, we reach our base case of 1 element.
Thus, $\left(\frac{k}{2^h}\right) = 1$, where h is the height of the recursion tree. To solve for h, it becomes $k = 2^h$, and to get h, we, it is simply $\log k$.

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.

First, we sort the array. This would take O($n \log n$) time. Then, we iterate through the array from 0 to n – 2, and we compare whether the i^{th} element is equal to the $(i+1)^{th}$ element. This would take O(n) time. Thus the time complexity would 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}.

First, we sort the array in ascending order, which would take O($n \log n$) time. Then, we iterate through the sorted array and check that it is not equal to the last element added to array B. If it is not the same, we can add the i^{th} element into array B. If they are the same, we simply ignore and iterate to the next element. Thus, the time complexity 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.

Sort both arrays A and B in ascending order. We can then make use of the merge step in Merge Sort to create array C. If the element to be added already exists in array C, we discard it and increment the pointer of array A or B. Thus, the time complexity 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.

Sort the array in ascending order. Then, we use two pointers to mark the beginning and ending of the array. If A[begin] + A[end] is greater than the target value, we decrement end by 1. If A[begin] + A[end] is less than the target value, we increment begin by 1. The time complexity for this algorithm is $O(n \log n)$.

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 threeyear-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.

We can come up with an algorithm similar to Merge Sort using the shoes as pivot.

1. Choose a random pair of shoes as the pivot.
2. Then, get every child to try on the pair of pivot shoes and group them into smaller and larger groups.
    a. If the shoes were too large for the toddler, place the toddler into the smaller group
    b. If the shoes were too tight and could not fit the toddler, place the toddler into the larger group
    c. If the shoes were an exact fit, it belongs to the toddler. This toddler would then be used as the pivot.
3. Once we have grouped the toddlers into larger and smaller groups and have found the pivot toddler, we can get the pivot toddler to try on the other pairs of shoes and group them into smaller and larger groups as well

a. If the shoes were too big on the pivot toddler, place the shoes into the larger group
b. If the shoes were too tight for the pivot toddler, place the shoes into the smaller group

4. We can then recurse on the two groups with the smaller group of toddlers trying on the smaller group of shoes and likewise for the larger group of toddlers trying on the larger group of shoes.

The time complexity for this algorithm would be $O(n \log n)$, 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?

The partitioning algorithm would partition the array into (k + 1) partitions. Then we can traverse the array, and place the elements into their respective partitions, such that they are less than or equals to pivot 1 (element <= pivot 1), larger than or equals to pivot k (element >= pivot k) or in between pivot i and pivot (i + 1) [pivot i <= element <= pivot (i + 1)]. Then, we proceed to recurse into the (k + 1) partitions and sort the elements in the subarrays.

(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*.

The time taken to sort and find perfect pivots would be $O(k \log k)$. The time taken to place the elements in their respective partition would be $O(n \log k)$ (via binary search). Thus, the asymptotic running time would be $O(k \log k) + O(n \log k)$.

(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*.

Since QuickSort partitions on k partitions with $\frac{n}{k}$ elements in each partition, and the partitioning cost is $O(n \log k)$, thus

$$T(n) = kT\left(\frac{n}{k}\right) + O(n \log k)$$

(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?

$$T(n) = kT\left(\frac{n}{k}\right) + O(n \log k)$$
$$=$$

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.)

We can have a left and right pointer that points to the start and end of the array. If the left pointer value is a 0, we keep incrementing the left pointer by 1 until it is pointing to a value of 1. Then, we proceed to look at the right pointer. Similarly, if the right pointer value is a 1, we will decrement the right pointer by 1, until it points to a value of 0. Then, we can proceed to do a swap with the left pointer's 1 and the right pointer's 0. This would put the previously right pointer's 0 in the left pointer's index and the previously left pointer's 1 in the right pointer's index. Then the process continues until the left pointer is no longer smaller than the right pointer's value. The running time would be O(n).

This can be done in place but it will not be stable. The algorithm can be made stable by using an extra array but this would incur additional space.

(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.

(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.

(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?