CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 5

For: February 7–February 11

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?

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

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

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

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

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

(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 $\{1, 2, 3\}$.

(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.

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

Problem 3. More 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.

Problem 5. Integer Sort

(a) Consider an array consisting of 0s and 1s, what is the most efficient way to sort it? Can you do this in-place? Is it 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.)

(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, do not try 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 place the output in.

(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 '1xxxxxxx' will come before all the elements of the (binary) form '0xxxxxxx'.

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.

(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?

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

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?

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

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

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

a) Yes it is possible e.g. [8 3 2 1 5 4 6 7 9]

b) No they are not stable. Yes it is possible, at the cost of O(n) additional space

```
def quickSort(arr):
    if len(arr) <= 1: return arr
    pivot = arr[-1]
    small, big = [], []
    for (index, value) in enumerate(arr):
        if index != len(arr)-1:
            if value <= pivot: small.append(value)
            else: big.append(value)
    return quicksort(small) + [pivot] + quicksort(big)
```

c) (i) $\Theta(n)$, only a single partition is needed

(ii) $O(nk)$

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

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

(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 $\{1, 2, 3\}$.

(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.

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

Problem 3. More Pivots! Yes

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) $O(n \log n)$

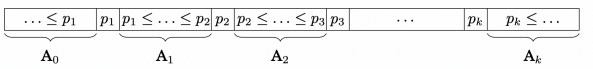

Fig. 1. Result of the partition step in k-pivot quicksort using pivots $p_1, \ldots, p_k$.

Note: A naive implementation for this partitioning step would be $O(nk)$.

$O(k \log k)$ to sort pivots

$O(n \log k)$ to place each item in the right bucket (using binary search)

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

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.

Double partition $\Theta(n \log n)$

$\Theta(1)$ 1. Pick a random child to be the pivot child

$\Theta(n)$ 2. Using the pivot child, linear scan the shoes to partition shoes into "too small" & "too big" (also note the matching shoe as the pivot shoe)

$\Theta(n)$ 3. Using the pivot shoe, linear scan to partition the children into "too small" & "too big" groups

$\Theta(1)$ 4. Remove pivot child & shoe from the overall group

$\Theta(\log n)$ 5. Recurse on the partitions

Problem 5. Integer Sort

(a) Consider an array consisting of 0s and 1s, what is the most efficient way to sort it? Can you do this in-place? Is it 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.)

(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, do not try 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 place the output in.

(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 '1xxxxxxx' will come before all the elements of the (binary) form '0xxxxxxx'.

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.

(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?

c) $64n = n \log n$

$\log n = 64$

$n = 2^{64}$

It is faster when $n > 2^{64}$

d) Divide each integer into 8 chunks of 8 bits each

Running time is now 2^8

Trade-off is $O(n)$ space