CS2040S Tutorial 3

Group T40

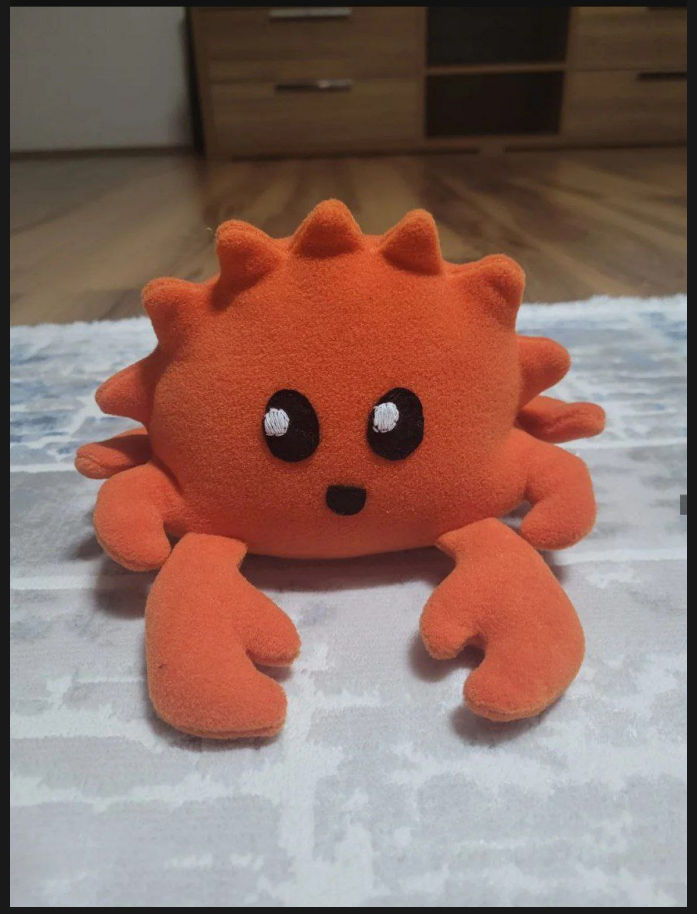
Week 5

Admin

- For future problem sets, make sure that your solution passes all private test cases.
  - If you didn't, then most likely your solution is not correct
  - If you believe there is a mistake in the private test case, let me know
- You may request for extension before the deadline of the PS

Picture of the Day

[media] My mom made this Ferris and I love him



PS3 Discussion

Problem 1: QuickSort Review

Problem 1a

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

Solution 1a

- Work one-by-one
- For first iteration, what should be the value of the first, middle and last key so it produces worst running time?
- After execution, repeat the same thing

Solution 1a (cont.)

- Suppose we have array of `0` to `9`
- `[a1, a2, a3, a4, a5, a6, a7, a8, a9]`
  - Median of `a1`, `a5`, `a9`
  - Worst case is when median is `8` (`9` is impossible, why?)
  - Then set `a1` = `8`, and `a9` = `9` (`a5` can set any values smaller than 8)
  - `[8, a2, a3, a4, a5, a6, a7, a8, 9]`
- `[a8, a2, a3, a4, a5, a6, a7, | 8, 9]`
  - Repeat, we have `a8` = `6`, `a7` = `7`
- Repeat and you will get `[8, 1, 3, 2, 5, 4, 7, 6, 9]` is one of the malicious test case
- Code it out :)

Problem 1b

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

Solution 1b

- In-place partitioning is not stable
- Possible to have stable QuickSort
  - Introduce auxilliary array that corresponds to the index of each element
  - If value of the array is the same, the corresponding value in the auxilliary array will be the tiebreaker
  - Note that swapping will swap element in the auxilliary array as well
  - Making the elements as pair is also possible :)

Problem 1c

Consider a QuickSort implementation that uses the 3-way partitioning scheme (i.e. elements equal to the pivot are partitioned into their own segment).

Problem 1c (i)

If an input array of size n contains all identical keys, what is the asymptotic bound for QuickSort?

Solution 1c (i)

$\mathcal{O}(n)$. Partition one time, all sorted :)

Problem 1c (ii)

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

For example, with $n = 6$ and $k = 3$, sort the array `[a, b, c, a, b, c]`

Solution 1c (ii)

- Observation: all elements of the same value as the pivot will not be recursed further
- The number of recursion is related to the number of distinct elements!
- Height of tree is bounded by $\mathcal{O}(k)$
- Each level takes $\mathcal{O}(n)$
- Hence, the asymptotic bound for the running time is $\mathcal{O}(nk)$

Solution 1c (ii)

- Assume pivot chosen is ideal, i.e. parititoned the array into two parts, where each part has almost the same number of ***distinct*** elements
- Recursion tree's height will be $\mathcal{O}(\lg k)$, each level takes $\mathcal{O}(n)$
- Time complexity: $\mathcal{O}(n \lg k)$

Solution 1c (ii) (cont.)

For those who want more rigourous proof, let $T(n, k)$ be the running time of quicksort to sort an array of size n with k distinct elements.

$$T(n,k) = \begin{cases} 1, & \text{if } k \leq 0 \\ T(n_1, k/2) + T(n_2, k/2) + \mathcal{O}(n), & \text{otherwise} \end{cases}$$

Where n_1 and n_2 are the numbers of elements for each subpart. Notice that $n_1 + n_2 < n$.

Solving this yield the same conclusion as previous slide

Problem 2: Array Processing Puzzles :o

Problem 2

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.

Solution

See the answer sheet

Problem 3: Child Jumble

Problem 3

- Twenty toddlers that want to find their shoes
- A pile of shoes
- Cannot compare shoe to shoe and feet to feet (indistinguishable by eyes)
- Try shoes on a toddler, and you know whether it fits, too large, or too small
- Design an algo + state time complexity

Solution 3

- QuickSort :)
- Choose any random shoes
  - For all toddlers, try the shoes. Partition them whose feet is smaller and bigger (obviously the matched size will be the pivot)
  - For all shoes, try it to the pivot toddler. Partition them into too-big shoes and too-small shoes for the toddler.
- Repeat until all is matched
- Time complexity: $\mathcal{O}(n \lg n)$, where n is the number of toddlers

Problem 4: More Pivots!

Problem 4

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.

Problem 4a

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?

Problem 4b

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.

Problem 4c

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 4

- Each element needs to know which partition it belongs to
- Given k pivots, we can sort it first in $\mathcal{O}(k \lg k)$ (e.g. using merge sort)
- For each element, we can binary search on the sorted k pivots to find where it belongs to
- Partition all elements takes $\mathcal{O}(n \lg k)$ time
  - Can extend the general idea of 3-way partition, by keeping $k + 1$ pointers ??? (how?)
  - Use auxilliary array to help (how?)

Solution 4 (cont.)

- Assuming that the runtime to sort an array of length n is $T(n)$
- $T(n) = (k+1) \cdot T(\frac{n}{k+1}) + \mathcal{O}(n \lg k)$
- Solving this yield $\mathcal{O}(n \lg k \log_k n)$
  - Tree has height of $\mathcal{O}(\log_k n)$ (assume partitioned evenly)
  - Each level takes $\mathcal{O}(n \lg k)$ time
- can be simplified to $\mathcal{O}(n \lg n)$
  - Any improvement?
  - More overhead
  - Perform better?
    - 2 and 3 pivots runs faster than ordinary quicksort :o

Problem 5: Integer Sort

Problem 5a

Given an array consisting of only 0’s and 1’s, what is the most efficient way to sort it?

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.)

Problem 5b

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.

Problem 5c

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

- First, use in-place algorithm from part (a) to sort the (first) highest order bit
  - At this stage, we have partitioned `1xxxxx` with `0xxxxx`
- Next, sort the second highest order bit using the same way for each part
- Repeat until the lowest order bit

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

Problem 5d

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