COMP 285 (NC A&T, Spr '22) Homework 3

Due. Tuesday, February 9th, 2022 @ 11:59 PM!

Homework Expectations: Please see Homework.

Exercises The following questions are exercises. We encourage you to work with a group and discuss solutions to make sure you understand the material.
Points This assignment is graded out of 100 points. However, you can get up to 120 points if you complete everything. These are not bonus points, but rather points to help make-up any parts you miss.

Fun with Divide and Conquer, Sorting, and Medians

Written Problems The following questions are to be submitted in written/typed form to gradescope.

1 Getting to Know You (10 pt.)

As part of this homework assignment, I'd like to get to know each of you.

Do the following:

- **(5 pt.)** Schedule a 10 minute conversation with me in the available slots here.
- **(5 pt.)** When you're done with the homework, complete this form.

[We are expecting: You should have scheduled a meeting with me for an available slot and submit the form. All of this is tracked and this is how we'll confirm to give you credit for this question.**]**

2 Exercise: A faster MergeSort? (10 pt.)

Inspired by all of our discussion on recursion and sorting, Miah, Ryan, and Jordan get together to try and figure out a faster[1] way to sort! After many attempts, they write down the following sorting function:

```
// MergeSort algorithm!
std::vector<int> mergeSortN(const std::vector<int>& input) {
    const std::size_t n = input.size();
    if (n <= 1) {
        return input;
    }
    std::vector<std::vector<int>> sortedSubvectors = {};
    for (int i = 0; i < n; i++) {
        std::vector<int> temp = mergeSortN(
            {input.begin() + i, input.begin() + i + 1}
        );
        sortedSubvectors.push_back(temp);
    }
    std::vector<int> sortedA = {};
    for (const std::vector<int> subVector : sortedSubvectors) {
        sortedA = merge(sortedA, subVector);
    }
    return sortedA;
}
```

2.1 What is this doing!? (5 pt.)

In your own words, describe that the algorithm above is doing. This should not take more than a few sentences, but should highlight how this is different from the normal implementation of `MergeSort`.

[We are expecting: A few sentences describing, in plain English, what the algorithm above does.**]**

2.2 How fast is it?! (5 pt.)

Jordan claims this version of `MergeSort` is much faster than any versions we've seen before. This is his argument:

[1] We will not cover in class, but it has been proven that there is no deterministic algorithm faster than $\theta(n \log n)$ for sorting numbers using comparisons.

This modified `MergeSort` splits the array into n subproblems of size $O(1)$ immediately. We therefore don't waste time with the '$log(n)$ levels' worth of splitting. Additionally, in this modified sort, we're calling `Merge` on a bunch of subvectors of size 1. Each merge would therefore take time

$$O(\text{size of subvector A}) + O(\text{size of subvector B}) = O(1) + O(1) = O(1)$$

That's contant time per merge! Yay! We therefore have an algorithm that sorts the input array in $O(n)$ time - n merges of $O(1)$ each!

Sadly, Jordan's analysis is **wrong**. It was a valiant try, though. Let's help Miah, Ryan, and Jordan out! What mistake are they making in their analysis above, and what is the true runtime of their modified `MergeSort`.

[We are expecting: An explanation (one sentence) that summarizes the mistake in the above reasoning and the correct runtime of `mergeSortN` along with an explanation (another sentence).**]**

3 Interview Practice: Finding Matching Elements (10 pt.)

Caleb is working on a "search" algorithm at Foogle. He is given an array `input` of n integers that is sorted in ascending order and contains only unique elements. That is to say, if his array is

$$a = [a_1, a_2, \cdots, a_{n-1}, a_n]$$

then $a_1 < a_2 < \cdots < a_{n-1} < a_n$.

His project is to search the above array and find the element such that its index is equal to its value, or returns that such i does not exist. That is to say, the algorithm finds i such that $i = a_i$.

3.1 A first proposal. (5 pt.)

Caleb proposes the following pseudo-code for an algorithm that solves the problem described above.

```
Algorithm:
    Input: A vector "input" sorted in ascending order with no duplicates.
    Outputs: The index i such that the i-th index contains the value i.

    n = len(input)
    for i = 0 to i = n - 1
        if (i == input[i]):
            return i

    throw "No such index exists"
```

Describe in plain English how Caleb's algorithm works, and provide the big-Oh running time.

[We are expecting: A few sentences describing how the algorithm Caleb wrote works in plain English, as well as $O(\cdots)$ where $\cdots$ is filled in with the proper running time.**]**

3.2 Your Proposal. (5 pt.)

After looking at Caleb's proposed algorithm, you realize that he's not taking advantage of the fact that the input array is sorted. Give an $O(\log n)$ algorithm that solves the same problem (eg, it finds the index i such that $i = a_i$ or returns that such i does not exist). If there are multiple such is, your algorithm can return any of them [2].

[We are expecting: Pseudocode for you algorithm, similar to what Caleb wrote in Part 3.1, as well as a plain English description of what your algorithm is doing and why it is correct. You **do not** need to proof correcness.**]**

[2] Consider a divide-and-conquer approach, similar to binary search! If you'd like to refresh your memory on binary search, see this video.

4 Interview Practice: Finding Medians (30 pt.)

You are given two arrays, each of length n. Your goal is to find the median of all elements of the two arrays.

4.1 Unsorted Arrays (10 pt.)

If the arrays **are not sorted**, give a $\Theta(n \log n)$ algorithm that returns the median of the combined arrays.

[We are expecting: A short English description of your approach, the pseudocode (similar to Part 3.2), and an explanation of your algorithm's runtime.**]**.

4.2 Unsorted Arrays But Faster (10 pt.)

If the arrays **are not sorted**, give an $\Theta(n)$ algorithm that returns the median of the combined arrays[3].

[We are expecting: A short English description of your approach, the pseudocode (similar to Part 3.2), and an explanation of your algorithm's runtime.**]**.

4.3 Sorted Arrays (10 pt.)

If the arrays **are sorted**, give an $O(\log n)$ algorithm that returns the median of the combined arrays. Here are a few hints to help you. Try to look at these only after you've given the problem a bit of thought on your own.

Hint 1 You probably want to use divide-and-conquer.

Hint 2 Calculating the median of a single sorted array can be done in $O(1)$ time (how?).

Hint 3 If the biggest element in one array (eg, the last element) is smaller than the smallest element in another array (eg, the first element), you can also compute the median $O(1)$ time (how?).

Hint 4 If you look at the medians of the two arrays, how does that help you split the problem into parts? What do you know if the median of the two arrays are equal? What if one is bigger than the other?

[We are expecting: A short English description of your approach, the pseudocode (similar to Part 3.2), and an explanation of your algorithm's runtime.**]**.

[3]Hint: You can use 'Select(A, k)' directly, without explanation, since this was given in lecture

5 Exercise: Randomized Algorithms (10 pt.)

In this exercise, we'll explore different types of randomized algorithms. We say that a randomized algorithm is a **Las Vegas algorithm** if it is always correct (that is, it returns the right answer with probability 1), but the running time is a random variable. We say that a randomized algorithm is a **Monte Carlo algorithm** if there is some probability that it is incorrect. For example, QuickSort (with a random pivot) is a Las Vegas algorithm, since it always returns a sorted array, but it might be slow if we get very unlucky.

We will visit the Majority Element problem to get more insight on randomized algorithms. The Majority Element problem is to find the element in a collection of elements that occurs at least half of the time. See Algorithm 4 for how we would check if an element is the majority.

We will assume that such an element always exists.

```
// Returns 2, because it occurs 4 times which is greater
// than 6/2 = 3.
findMajorityElement({2, 2, 2, 2, 3, 5});
// Returns 10, because it occurs 3 times which is greater
// than 5/2 = 2.5
findMajorityElement({10, 10, 10, 2, 4});
// Returns 4 because it occurs 3 times, which is greater
// than 4/2 = 2.
findMajorityElement({4, 4, 2, 4})
```

Algorithm	Monte Carlo or Las Vegas?	Expected running time	Worst-case running time	Probability of returning a majority element
Algorithm 1				
Algorithm 2				
Algorithm 3				

[We are expecting: Your filled in-table. You may use asymptotic notation for the running times.**]**

Algorithm 1: findMajorityElement1

Input: A population P of n elements
while *true* **do**
    Choose a random $p \in P$;
    if isMajority*(P, p)* **then**
        return p;

Algorithm 2: findMajorityElement2

Input: A population P of n elements
for *100 iterations* **do**
    Choose a random $p \in P$;
    if isMajority*(P, p)* **then**
        return p;
return $P[0]$*;*

Algorithm 3: findMajorityElement3

Input: A population P of n elements
Put the elements in P in a random order.;
`/* Assume it takes time` $\Theta(n)$ `to put the` n `elements in a random order */`
for $p \in P$ **do**
    if isMajority*(P, p)* **then**
        return p;

Algorithm 4: isMajority

Input: A population P of n elements and a element $p \in P$
Output: True if p is a member of a majority species
count $\leftarrow 0$;
for $q \in P$ **do**
    if $p = q$ **then**
        count ++;
if $count > n/2$ **then**
    return *True*;
else
    return *False*;

Coding Problems The following questions are to be submitted as a ".zip" file on Gradescope.

6 Coding (50 pt.)

(50 pt.) After completing the written portion of the assignment, you should submit to Gradescope.

You can get your starter code for the coding portion here.

Note that the starter code also include a few test cases you can run on repl.it. However, the full test suite is the one run on Gradescope.

Please reference the `README.md` included in your starter code for detailed instructions.

Submitting the Assignment

This assignment is a combination of written and programming questions. Both portions of the assignment should be submitted through Gradescope.

The "Homework 3: Fun with Divide and Conquer, Sorting, and Medians" assignment is the written portion, for which you should submit a **typed** response to the non-coding questions (questions 1-5). Each response should clearly be marked with its corresponding number. You are free to use the provided templates, print the questions and write your answers, or to simply type your responses on a blank document (whatever works for you).

The "Homework 3: Coding" is the programming portion of the assignment. For this portion, download the ".zip" file from replit and upload this ".zip" file as your answer to Gradescope. You can upload the assignment as many times as you want.