**COMP 285 (NC A&T, Spr ‘22)Homework 3**

#### Due.

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

#### Homework Expectations:

Please see [Homework](https://www.comp285.ml/homework/#general-homework-information).

#### 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.

# Getting to Know You

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

Do the following:

* Schedule a 10 minute conversation with me in the available slots [here](https://calendly.com/comp285-prof/get-to-know-you?month=2022-02).
* When you’re done with the homework, complete this [form](https://forms.gle/KpC8RJ3gvkPgQPYY7).

# Exercise: A faster MergeSort?

Inspired by all of our discussion on recursion and sorting, Miah, Ryan, and Jordan get together to try and figure out a faster[[1]](#footnote-30) 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;
}

## What is this doing!?

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.

## How fast is it?!

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

This modified MergeSort splits the array into $n$ subproblems of size $O\left(1\right)$ immediately. We therefore don’t waste time with the ‘$log\left(n\right)$ 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

$$O\left(size of subvector A\right)+O\left(size of subvector B\right)=O\left(1\right)+O\left(1\right)=O\left(1\right)$$

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

# Interview Practice: Finding Matching Elements

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

$$a=\left[a\_{1},a\_{2},\cdots ,a\_{n−1},a\_{n}\right]$$

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 $i=a\_{i}$.

## A first proposal.

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.

## Your Proposal.

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 $O\left(logn\right)$ 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 $i$s, your algorithm can return any of them [[2]](#footnote-35).

# Interview Practice: Finding Medians

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

## Unsorted Arrays

If the arrays **are not sorted**, give a $Θ\left(nlogn\right)$ algorithm that returns the median of the combined arrays.

.

## Unsorted Arrays But Faster

If the arrays **are not sorted**, give an $Θ\left(n\right)$ algorithm that returns the median of the combined arrays[[3]](#footnote-40).

.

## Sorted Arrays

If the arrays **are sorted**, give an $O\left(logn\right)$ 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.

1. You probably want to use divide-and-conquer.
2. Calculating the median of a single sorted array can be done in $O\left(1\right)$ time (how?).
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 $O\left(1\right)$ time (how?).
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?

.

# Exercise: Randomized Algorithms

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 [[alg:isMajority]](#alg:isMajority) 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** |  |  |  |  |

 Choose a random $p\in P$

 Choose a random $p\in P$

Put the elements in $P$ in a random order.

count $\leftarrow $ 0

#### Coding Problems

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

# Coding

After completing the written portion of the assignment, you should submit to [Gradescope](https://www.gradescope.com/courses/350304).

You can get your starter code for the coding portion [here](https://replit.com/team/COMP285/HW3-Code).

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](https://www.gradescope.com/courses/350304).

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](#sec:last)). 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](https://www.gradescope.com/courses/350304). You can upload the assignment as many times as you want.

1. We will not cover in class, but it has been proven that there is no deterministic algorithm faster than $θ\left(nlogn\right)$ for sorting numbers using comparisons. [↑](#footnote-ref-30)
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](https://www.youtube.com/watch?v=P3YID7liBug). [↑](#footnote-ref-35)
3. Hint: You can use ‘Select(A, k)’ directly, without explanation, since this was given in lecture [↑](#footnote-ref-40)