CS 344 - Fall 2022
Homework 2.
100 points total plus 15 points extra credit

1 Problem 1 (30 points total)

For all the recursive formula problems below, you can assume that $T(1) = T(2) = 1$.

- **Part 1:** (15 points) Consider an algorithm that solves a problem of size n by dividing it into 6 pieces of size $n/6$, recursively solving each piece, and then combining the solutions in $O(n^3)$ time. What is the recurrence formula for this algorithm? Use a recursion tree to figure out the running time of the algorithm (in big-O notation, as always) AND then use a proof by induction to show that your result is correct.
- **Part 2:** (7 points) Say that you have an algorithm with recurrence formula $T(n) = 16T(n/4) + n^2$. Use a recursion tree to figure out the running time of your algorithm. *No need for an induction proof of this one.*
- **Part 3:** (8 points) Say that you have an algorithm with recurrence formula $T(n) = 8T(n/5) + n$. Use a recursion tree to figure out the running time of your algorithm. *No need for an induction proof on this one.*

2 Problem 2 (15 points total)

For this problem, we are assuming that arithmetic operations take $O(1)$ time. So addition, subtraction, division, and multiplication, are done in $O(1)$ time no matter how big the numbers are. (This is not exactly true in practice, but it's a reasonable approximation to reality.)

Now consider the following function SQRT(n)

- Input: a positive integer n
- Output: an integer k such that $k^2 = n$, or "No Solution" if none exists

Example: SQRT(9) should output 3, while SQRT(10) should output "no solution"

The Problem Write pseudocode for a recursive algorithm that computes SQRT(n) in time $O(\log(n))$. In addition to pseudocode, you should state the recursion formula for your algorithm. So all you need is pseudocode and the recursion formula, nothing else. You do NOT need to prove that this recursion formula solves to $T(n) = O(\log(n))$

NOTE Don't use tools from outside this class. For example, some of you know Newton's method, but you should not use that. I'm looking for a simple algorithm that only uses ideas we've covered in class so far.

3 Problem 3 (20 points)

Given an array A, we say that elements $A[i]$ and $A[j]$ are swapped if $j > i$ but $A[j] < A[i]$.

Example 1: If $A = [8, 5, 9, 7]$, then there are a total of 3 swapped pairs, namely 8 and 5; 8 and 7; and 9 and 7.

Example 2: if $A = 2, 3, 4, 5, 1, 6, 7, 8$ then there are a total of 4 swapped pairs: 1 and 2; 1 and 5; 1 and 3; 1 and 4.

The goal for this problem is to write a recursive algorithm that given an array A determines the number of swapped pairs in the array in $O(n\log(n))$ time. The algorithm to do this is extremely similar to merge sort; in fact, the algorithm does all of merge sort (verbatim), plus a few extra things to keep track of the number of swaps. In particular, one side effect of the algorithm is to sort the array A.

what you need to do: In order to save you the trouble of retyping the pseudocode for merge sort,I have written the pseudocode for most of the algorithm. Your job is simply to fill in two lines. These are the key lines that keep track of the number of swaps.

Note that the algorithm below has an outer function MergeSortAndCountSwaps which I wrote entirely; nothing to fill in. But MergeSortAndCountSwaps then calls MergeAndCountSwapsBetween as a subroutine; in that subroutine you will have to fill in two lines.

The Algorithm MergeSortAndCountSwaps(A)
INPUT: array A with unique elements (no duplicates)
OUTPUT: pair (SortedA, S), where SortedA is the array A sorted in increasing order and S is the total number of swapped pairs in A.

```
• A1 ← first half of A. So $A1 = A[0], ..., A[\frac{n}{2} - 1]$
• A2 ← second half of A. So $A2 = A[\frac{n}{2}], ..., A[n-1]$
• (SortedA1,$S_1$) ← MergeSortAndCountSwaps(A1)
• (SortedA2,$S_2$) ← MergeSortAndCountSwaps(A2)
• (SortedA, $S_3$) ← MergeAndCountSwapsBetween(SortedA1,SortedA2)
• Return (SortedA, $S_1 + S_2 + S_3$).
```

We now need to define MergeAndCountSwapsBetween(A, B). This is the part where I will have you fill in a few lines. Note that MergeAndCountSwapsBetween(A, B) is very similar to Merge(A,B) from class, and in particular it is NOT a recursive function.

MergeAndCountSwapsBetween(A,B):

```
1. Initialize array $C$ of size $2n$
2. $i \leftarrow 0$
3. $j \leftarrow 0$
4. $S' \leftarrow 0$        Comment: $S'$ will count swaps; the answer lines below
   which you have to add will be responsible for changing $S'$.
5. While (i < n OR j < n)
   (a) if $j = n$ or A[i] < B[j] :
         • **YOUR ANSWER 1 GOES HERE**
         • set next element of $C$ to $A[i]$
         • (formally, $C[i + j] \leftarrow A[i]$)
         • i ← i+1
   (b) if i = n or A[i] > B[j]:
         • **YOUR ANSWER 2 GOES HERE**
```

- set next element of C to $B[j]$
- j $\leftarrow$ j+1

6. output pair (C, S').

Questions you need to answer:

- What goes in Answer 1? *your answer should be a single line.*
- What goes in Answer 2? *your answer should be a single line.*

NOTE: please don't recopy all of the pseudocode above. Your answer to this problem should just be:

- Answer Line 1: (your answer here)
- Answer Line 2: (your answer here)

4 Problem 4 (35 points total)

Given an array A of length n, we say that x is a *frequent* element of A if x appears at least $n/4$ times in A. Note that an array can have anywhere between 0 and 4 frequent elements.

Now, Say that your array A consists of incomparable objects, where you can ask whether two objects are equal (i.e. check if $A[i] == A[j]$), but there is NOT a notion of one object being bigger than another. In particular, we cannot sort A or find the median of A. (Think of e.g. an array of colors.)

4.1 Part 1 (5 points)

Write pseudocode for an algorithm IsFrequent(A,x) which takes as input the array A of incomporable objects and a single element x, and returns TRUE if x is frequent in A and FALSE otherwise. Your algorithm should run in $O(n)$ time, where n is the length of array A.

NOTE: you may NOT use a dictionary/hash-function for this problem.

HINT: this one is very short and does not require recursion.

4.2 Part 2 (30 points)

Given an array A of n incomparable objects, write pseudocode for an agorithm that finds all the frequent elements of A, or returns "no solution" if none exists. Your algorithm must be recursive and should run in $O(n \log(n))$ time. *In addition to the pseudocode, you must state the recurrence formula for the algorithm. You do need an explanation; you just need to state what the recurrence formula is.*

HINT: your pseudocode will be simpler if you use the algorithm IsFrequent from part 1.

NOTE: don't try anything silly like converting the incomparable objects to integers so that you can then sort the integers. I mean it when I say no sorting, no selection!

NOTE: you may NOT use a dictionary/hash-function for this problem.

NOTE: For this problem you MUST use a recursive algorithm. This is for your own sake, since the non-recursive algorithm for this problem is *much* more complicated, and will take you down the wrong track. In particular, don't try to modify Moore's majority algorithm, as that is not recursive. (If you've never heard of Moore's majority algorithm, don't worry about it; it's not relevant to this problem.)

5 Problem 5 – Extra Credit (15 points)

Give an $O(n)$ algorithm for problem 4. *Your algorithm does not need to be recursive.*

For this problem, you need to justify both correctness and running time. As always, "justify" means that a reader who doesn't know the solution should be able to read your solution and be convinced that it is correct.

IMPORTANT NOTE: you may NOT use a dictionary/hash-function for this problem. Recall also that the elements of the array are incomparable, so you may not use sorting/selection either.

NOTE: A good starting point for this extra credit problem is Moore's majority algorithm. If you haven't seen it before, you can read about it at this webiste:

https://www.geeksforgeeks.org/boyer-moore-majority-voting-algorithm/

But note that here we are looking for frequent elements rather than simply the majority, and this will require you to significantly modify Moore's majority algorithm. The proof that the algorithm works is more complex than for Moore's majority algorithm.

FULL DISCLOSURE: This one is quite tricky, even by extra credit standard.