CS 161 (Stanford, Winter 2022) Homework 2

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards. You should cite all sources you used outside of the course material.
What we expect: Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Exercises. The following questions are exercises. We suggest you do these on your own. As with any homework question, though, you may ask the course staff for help.

1 Exercise: Recurrence Relations!

Calculate an upper-bound runtime for each of the following recurrence relations. Use any method we've seen in class—Master Theorem, Substitution Method, algebra, etc. (You may also use a calculator if needed!)

(a) **(2 pt.)** $T(n) = 18T(n/9) + O(n\sqrt{n})$, where $T(1) = 1$.

(b) **(2 pt.)** $T(n) = T(n-1) + 1$, where $T(1) = 1$.

(c) **(3 pt.)** $T(n) = T(\sqrt{n}) + 2^{32}$ where $T(x) = 1, \forall x \leq 2$. (Hint: Write n as a power of two).

[We are expecting: Use of Master Theorem or Substitution Method to calculate the tightest upper bound for each of these recurrence relations in big-O notation.**]**

2 Exercise: Modified MergeSort

Let's see how changing the size of the subproblems affects MergeSort.

(a) **(3 pt.)** Your friend gives you this modified version of MergeSort, and he claims that it runs asymptotically better than the version of MergeSort we showed in class. Is he correct in his claim? Write down a recurrence relation and runtime for this version of MergeSort. (`Merge` is the same as we saw in lecture)

```
MergeSortThirds(A):
  n = len(A)
  if(n <= 1):
    return A

  L = MergeSortThirds(A[:n/3])
  M = MergeSortThirds(A[n/3:2n/3])
  R = MergeSortThirds(A[2n/3:])

  temp = Merge(L,M)
  return Merge(temp,R)
```

(b) **(4 pt.)** Inspired by your friend's idea, Pepper the Peculiar Penguin (Plucky's little sibling) decides to write the following version of MergeSort:

```
MergeSortN(A):
  n = len(A)
  if n <= 1:
      return A

  sortedSublists = []
  for i = 0,...,n-1
      temp = MergeSortN(A[i:i+1] )
      sortedSublists.append(temp)


  sortedA = []
  for each sublist in sortedSublists:
    sortedA = Merge(sortedA, sublist)

  return sortedA
```

Pepper says that this version of MergeSort is faster than the version we saw in lecture. Pepper's argument is as follows:

> "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 sublists of size 1. Each merge would therefore take time
>
> $$O(\text{size of sublist A}) + O(\text{size of sublist B}) = O(1) + O(1) = O(1)$$
>
> That's constant time per merge! Yay! We now have an algorithm that should sort the array A in $O(n)$ time — n merges of $O(1)$ time each."

Sadly, Pepper's analysis is wrong. It was a good try, though!

Let's help a penguin out: Tell Pepper what her mistake was and explain what the true runtime of this modified MergeSort truly is.

[We are expecting: For part a), a recurrence relation for `MergesortThirds` and and explanation of its runtime. For part b), an explanation of Pepper's incorrect reasoning and an explanation of the true runtime of `MergesortN`**]**

3 Index Match

(8 pt.) You are given an array a of n integers where $a_1 < a_2 < ... < a_n$. Give an $O(\log n)$ algorithm that finds an index i where $a_i = i$ or returns that such i does not exist. If there are multiple such i's, your algorithm can return any of these.
[We are expecting: Pseudocode for your algorithm and a clear English description of what your algorithm is doing and why it is correct. You do not need to prove that your algorithm is correct**]**

Problems. The following questions are problems. You may talk with your fellow CS 161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

4 Majority Vote

The magical country of Algorithmia recently held an election. Every vote cast was electronic, but unfortunately, a recent power surge caused a malfunction in the system just before votes were counted. In fact, the system became so fried, that we only know the following facts about the election:

- There are exactly n citizens in Algorithmia, and all of them voted.
- Exactly one candidate received the majority vote. In other words, some candidate received strictly greater than n/2 votes.
- We don't know how many candidates there were.

Algorithmia's government has heard about your algorithmic expertise, since you took CS 161. As a result, they've hired you to count the votes. However, because of the malfunction, you have limited resources. Algorithmia's cybersecurity software is only allowing you to open a single ballot. As a result, you must find a ballot that voted for the winning candidate. The only function you have unlimited access to is:

$$\texttt{isMatching}(\text{ballotA}, \text{ballotB})$$

This function returns True if ballotA and ballotB cast a vote for the same candidate, and False otherwise.

(a) **(7 pt.)** Design a deterministic divide-and-conquer algorithm which uses $O(n \log n)$ calls to `isMatching` and returns the winning candidate.
[We are expecting: Pseudocode that calls `isMatching` AND a clear English description of what your algorithm is doing.**]**

(b) **(3 pt.)** Explain why your algorithm calls `isMatching` $O(n \log n)$ times.
[We are expecting: A short English justification of the number of calls to `isMatching`. You may use Master Theorem if it applies. **]**

(c) **(10 pt.)** Use a proof by induction to show that your algorithm is correct.
[We are expecting: A rigorous proof by induction.**]**

(d) **(0 pt.)** BONUS (optional) Is $O(n \log n)$ the fastest algorithm you can come up with? Either give an asymptotically faster algorithm which finds a majority-voted candidate,

or else prove that no such algorithm exists.
[We are expecting: Nothing! This part is not required. You can provide an English description of your faster algorithm alongside an justification of its runtime OR a proof that $\Omega(n \log n)$ is indeed the lower bound for this problem.**]**

5 Embedded Ethics

When we work with algorithms, we often need to ignore or change aspects of a real world situation in order to turn it into an algorithmically solvable problem. For example, we can write an algorithm that sorts a numbered list without knowing what the numbers are numbers of.

- **Abstraction** is when we omit details of the real world situation.
- **Idealization** is when we deliberately change aspects of the real world situation.

(a) **(2 pt.)** Does the algorithmic problem solved by your majority-finding algorithm in Problem 4 make abstractions? What are they?
[We are expecting: 1-4 sentences which identify one or more aspects of the problem setup above as abstractions**]**

(b) **(3 pt.)** What specific problem does the algorithm in Problem 4 solve, and in what respects does it differ from the real-world problem of determining who should be the future ruler of Algorithmia?
[We are expecting: 3-4 sentences which identify the problem solved by the algorithm; the real-world problem; at least 1 difference and 1 similarity between them.**]**

(c) **(3 pt.)** The "Majority Vote" problem presents you with two idealizing assumptions, which are sometimes false in real-world situations, but make the problem algorithmically tractable. What are they? Would your "Find Majority" algorithm solve the real world problem if either of these assumptions did not hold true?
[We are expecting: 4-6 sentences which identify two aspects of the problem setup above as idealizations. For each, explain whether and how it would divert us from the real-world problem, if it failed to hold true.**]**

6 Quokka Island

You arrive on an island full of many quokkas. (You must be close to Australia!) On this island, there are nice quokkas and nasty quokkas. The nice quokkas always tell the truth; the nasty quokkas may lie or may tell the truth. The quokkas themselves can tell who is nice and who is nasty, but an outsider can't tell the difference: they all just look like quokkas.

You arrive on this island, and are tasked with finding the nice quokkas. You are allowed to pair up the quokkas and have them evaluate each other. For example, if Nancy the quokka and Ned the quokka are both Nice quokkas, then they will both say that the other is nice.

But if Nancy the quokka is a Nice quokka and Nefario the quokka is a Nasty quokka, then Nancy will call Nefario out as nasty, but Nefario may say either that Nancy is nasty or that she is nice. We will refer to one of these interactions as a "quokka-to-quokka comparison." The outcomes of comparing quokkas A and B are as follows:

quokka A	quokka B	A says (about B)	B says (about A)
Nice	Nice	Nice	Nice
Nice	Nasty	Nasty	Either
Nasty	Nice	Either	Nasty
Nasty	Nasty	Either	Either

Suppose that there are n quokkas on the island, and that there are strictly more than $n/2$ nice quokkas. In this problem, you will develop an algorithm to find all of the nice quokkas, that only uses $O(n)$ quokka-to-quokka comparisons. Before you start this problem, think about how you might do this—hopefully it's not at all obvious! Along the way, you will also practice some of the skills that we've seen in Week 1. You will design two algorithms, formally prove that one is correct using a proof by induction, and you will formally analyze the running time of a recursive algorithm.

(a) **(4 pt.)** Give an algorithm that uses $O(n^2)$ quokka-to-quokka comparisons and identifies all of the nice quokkas.
[We are expecting: A description of the procedure (either in pseudocode or very clear English), with a brief explanation of what it is doing and why it works.**]**

(b) **(12 pt.)** * Now let's start designing an improved algorithm. The following procedure will be a building block in our algorithm—make sure you read the requirements carefully! Suppose that n is even. Show that, using only $n/2$ quokka-to-quokka comparisons, you can reduce the problem to the same problem with less than half the size. That is, give a procedure that does the following:

- **Input:** A population of n quokkas, where n is even, so that there are strictly more than $n/2$ nice quokkas in the population.
- **Output:** A population of m quokkas, for $0 < m \leq n/2$, so that there are strictly more than $m/2$ nice quokkas in the population.
- **Constraint:** The number of quokka-to-quokka comparisons is no more than $n/2$.

[We are expecting: A description of this procedure (either in pseudocode or very clear English), and rigorous argument that it satisfies the **Input**, **Output**, and **Constraint** requirements above.**]**

(c) **(0 pt.) [This problem is NOT REQUIRED, but you may assume it for future parts. Note: the maximum assignment score will still be capped at 100.]** Extend your argument for odd n. That is, given a procedure that does the following:

*This is the trickiest part of the problem set! You may have to think a while.

- **Input:** A population of n quokkas, where n is odd, so that there are strictly more than $n/2$ nice quokkas in the population.
- **Output:** A population of m quokkas, for $0 < m \leq \lceil n/2 \rceil$, so that there are strictly more than $m/2$ nice quokkas in the population.
- **Constraint:** The number of quokka-to-quokka comparisons is no more than $\lfloor n/2 \rfloor$.

($\star$) *For all of the following parts, you may assume that the procedures in parts (b) and (c) exist even if you have not done those parts.*

(d) **(4 pt.)** Using the procedures from parts (b) and (c), design a recursive algorithm that uses $O(n)$ quokka-to-quokka comparisons and finds a *single* nice quokka.
[We are expecting: A description of the procedure (either in pseudocode or very clear English).**]**

(e) **(8 pt.)** Prove formally, using induction, that your answer to part (d) is correct.
[We are expecting: A formal argument by induction. Make sure you explicitly state the inductive hypothesis, base case, inductive step, and conclusion.**]**

(f) **(8 pt.)** Prove that the running time of your procedure in part (d) uses $O(n)$ quokka-to-quokka comparisons. If you find that you are working with floors and ceilings, you may ignore them (i.e. assume that the quantity is a whole number).
[We are expecting: A formal argument. Note: do this argument "from scratch," do not use the Master Theorem.**]**

(g) **(4 pt.)** Give a procedure to find *all* nice quokkas using $O(n)$ quokka-to-quokka comparisons.
[We are expecting: An informal description of the procedure. **]**

7 Matrix Multiplication

Suppose that we have two $n \times n$ matrices X and Y and we'd like to multiply them.

(a) **(2 pt.)** What is the running time of the standard algorithm (that computes the inner product of rows of X and columns of Y)? You can assume that simple arithmetic operations, like multiplication of numbers, take constant time ($O(1)$).
[We are expecting: A detailed analysis of the runtime of your algorithm including the Big-O time in terms of n.**]**

(b) **(3 pt.)** Now let's divide up the problem into smaller chunks like this, where the eight $\frac{n}{2} \times \frac{n}{2}$ sub-matrices (A, B, C, D, E, F, G, H) are each quarters of the original matrices, X and Y:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

We now have a divide and conquer strategy! Find the recurrence relation of this strategy and the runtime of this algorithm.

[We are expecting: A detailed analysis of the runtime of your algorithm including the Big-O time in terms of n.**]**

(c) **(3 pt.)** Can we do better? It turns out we can by calculating only 7 of the subproblems:

$$P_1 = A(F - H) \qquad P_5 = (A + D)(E + H)$$
$$P_2 = (A + B)H \qquad P_6 = (B - D)(G + H)$$
$$P_3 = (C + D)E \qquad P_7 = (A - C)(E + F)$$
$$P_4 = D(G - E)$$

And we can solve XY by

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

We now have a more efficient divide and conquer strategy! What is the recurrence relation of this strategy and what is the runtime of this algorithm? **[We are expecting:** A detailed analysis of the runtime of your algorithm including the Big-O time in terms of n.**]** Historical note: Sophisticated variants of this strategy have resulted in (asymptotically) better-and-better algorithms for matrix multiplication over the years. For a humorous take on the most recent improvement, see here (just for fun): https://www.smbc-comics.com/comic/mathematicians.

(d) **(2 pt.)** Your friend tried to solve part (c) of the problem, and came to the following conclusion:

Claim: The algorithm runs in time $T(n) = O(n^2 \log(n))$.

Proof: At the top level, we have 7 operations adding/subtracting $n/2 \times n/2$ matrices, which takes $O(n^2)$ time. At each subsequent level of the recursion, we increase the number of subproblems by a constant factor (7), and the subproblems only become smaller. Therefore, the running time per level can only increase by a constant factor. By induction, since for the top level it is $O(n^2)$, it will be $O(n^2)$ for all levels. There are $O(\log(n))$ levels, so in total the running time is $T(n) = O(n^2 \log(n))$.

What is the error in this reasoning?

[We are expecting: A clear and concise description, in plain English, of the error with this proof.**]**