Homework 4

Name:

Due October 6 (11:59 p.m.)

Instructions

- **Answers should be written in the boxes by modifying the provided Latex template or some other method answers are clearly marked and legible. Submit via Gradescope.**
- **Honors questions are optional. They will not count towards your grade in the course. However you are encouraged to submit your solutions to these problems to receive feedback on your attempts.**
- **You must enter the names of your collaborators or other sources (including use of LLMs) as a response to Question 0. Do NOT leave this blank; if you worked on the homework entirely on your own, please write "None" here. Even though collaborations in groups of up to 3 people are encouraged, you are required to write your own solution.**
- **For the late submission policy, see the website.**
- Acknowledgment - Thank you to Amir, Daji, Oded, Peter and Rotem for help in setting the problem set.

1-0 List all your collaborators and sources: ($-\infty$ points if left blank)

tutor, textbook, kids @ tutoring

Problem 4-1 – Permutations (40 Points)

Let $[n] := \{1, 2, \ldots, n\}$, and let S_n be the set of all permutations of $[n]$. For example, for $n = 3$, we have $|S_3| = 6$ with:

$$S_3 = \{(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)\}$$

Note that each permutation in S_n can be seen as an input for a sorting algorithm (where the elements of $[n]$ are reordered according to the permutation). In this question, we will consider sorting algorithms that for some inputs err and do not return the correct answer. We say that a sorting algorithm is ε-correct if the algorithm produces the correct result (i.e., produces a sorted array as output) on at least $\varepsilon \cdot |S_n|$ inputs in the set S_n.

1. (25 points) Argue that for any $0 \leq \varepsilon \leq 1$, the decision tree of an ε-correct comparison-based sorting algorithm must have at least $\varepsilon \cdot |S_n|$ leaves. (**Hint**: Recall that the output of a comparison-based sorting algorithm is a permutation that prescribes how the input array has to be ordered to obtain the sorted output.)

Solution: Let's say A is an ε-correct comparison algo. If A is in fact ε-correct, it's true that the A-algo is giving correct output for ε proportion of inputs. By contradiction, let's say that the decision tree for the A-algo has less than $\varepsilon * |S_n|$ leaves but that there are still $|S_n|$ inputs to the A-algo. But we're still going with the assumption that A is ε-correct, so it's going to produce the correct outputs for $\varepsilon * |S_n|$ inputs. For the contradiction, the number of leaves (which are the outputs) is less than the number of required inputs to maintain $\varepsilon * |S_n|$. As per the Pigeon Hole Principle, there's going to be at least one pair of two different inputs that'll be assoc. w/ the same output-permutation. But that output-permutation is only prescribing the correct re-ordering for one input, not two. So one of those inputs is assigned to an incorrect output, which violates the A-algo being ε-correct, proving that the decision tree needs at least $\varepsilon * |S_n|$ leaves to accommodate the outputs while still being correct.

In the following, we investigate whether allowing for more errors can save on the required number of comparisons. For example, at the extreme case that $\varepsilon = 0$, no comparison has to be made (the algorithm is allowed to always err). In what follows, for each given value of ε, decide whether there exists a comparison-based sorting algorithm that is ε-correct and runs in time $O(n)$. Justify your answers. (Recall that $|S_n| = n! = n(n-1)\cdots 2\cdot 1$.)

2. (5 points) $\varepsilon = 1/2$.

Solution: $\varepsilon = \frac{1}{2}$, meaning half of the outputs need to be correctly sorted.
$|S_n| \rightarrow n!$, meaning there are going to be $\varepsilon * n!$ leaves for all the combinations.
This results in $\varepsilon * n! = \frac{1}{2} * n! = \frac{n!}{2}$.
This represents the number of leaves that the tree needs to be ε correct. Even at the minimum number of leaves required, the algorithm won't be running in $O(n)$ time. This is because even the best sorting algorithm will go down the deepest branch of the tree, resulting in $log_2(n)$ levels. If it's going down $log_2(n)$ levels at a minimum with at least $\frac{n!}{2}$ leaves, then the runtime will be $log(\frac{n!}{2})$. We know that $n! > (\frac{n}{2})^{\frac{n}{2}}$.

$$\rightarrow \log(\frac{1}{2}n!) \rightarrow \log(\frac{1}{2}) + \log(n!) \rightarrow \log_2(2^{-1}) + \log(n!)$$
$$\rightarrow -1 + \log(n!)$$
$$\rightarrow \log(n!) - 1 > \log(\frac{n}{2})^{\frac{n}{2}} - 1 \text{ because we know that } n! > (\frac{n}{2})^{\frac{n}{2}}$$
$$\rightarrow \log(\frac{n}{2})^{\frac{n}{2}} \rightarrow \frac{n}{2}\log(\frac{n}{2}) - 1$$
$$\rightarrow \frac{n}{2}\log(\frac{n}{2}) - 1 \rightarrow \Omega(n\log(n))$$
$$\rightarrow \log(n!) - 1 = \Omega(n\log(n)) \text{ meaning LHS is already greater than fxn inside Omega}$$
$$\rightarrow \log(\varepsilon * n!) \neq O(n)$$

because nlogn already grows faster than n-time and LHS is greater than nlogn, so it can't be equal to O(n). If the LHS is larger than the best case (Omega) of $n\log(n)$, then there's no way it can be $O(n)$.

3. (5 points) $\varepsilon = 1/n$.

Solution: $\varepsilon = \frac{1}{n}$, meaning 1/n outputs need to be correctly sorted.
$|S_n| \to n!$, meaning there are going to be $\varepsilon * n!$ leaves for all the combinations.
This results in $\varepsilon * n! = \frac{1}{n} * n! = \frac{n!}{n}$.
This represents the number of leaves that the tree needs to be ε correct. Even at the minimum number of leaves required, the algorithm won't be running in $O(n)$ time. This is because even the best sorting algorithm will go down the deepest branch of the tree, resulting in $log_2(n)$ levels. If it's going down $log_2(n)$ levels at a minimum with at least $\frac{n!}{n}$ leaves, then the runtime will be $log(\frac{n!}{n})$.

$$\to \text{We know that } n! > (\frac{n}{2})^{\frac{n}{2}}$$
$$\to \log(n!) \geq \frac{n}{2}\log(\frac{n}{2}) = \frac{n}{2}(\log n - \log 2) = \frac{n}{2}\log n - \frac{n}{2}\log 2$$
$$\to \log\frac{n!}{n} = \log n! - \log n \geq \frac{n}{2}\log n - \frac{n}{2}\log n - \log n$$
$$\to \log\frac{n!}{n} \geq \frac{n}{2}\log n - O(n) = \Omega(n\log n)$$
$$\to \log(\varepsilon * n!) \neq O(n)$$

Because $\Omega(n\log n)$ grows faster than O(n), $\log\frac{n!}{n}$ cannot be O(n).

4. (5 points) $\varepsilon = \frac{1}{2^n}$.

Solution: $\varepsilon = \frac{1}{2^n}$, meaning $\frac{1}{2^n}$ outputs need to be correctly sorted.
$|S_n| \to n!$, meaning there are going to be $\varepsilon * n!$ leaves for all the combinations.
This results in $\varepsilon * n! = \frac{1}{2^n} * n! = \frac{n!}{2^n}$.
This represents the number of leaves that the tree needs to be ε correct. Even at the minimum number of leaves required, the algorithm won't be running in $O(n)$ time. This is because even the best sorting algorithm will go down the deepest branch of the tree, resulting in $log_2(n)$ levels. If it's going down $log_2(n)$ levels at a minimum with at least $\frac{n!}{2^n}$ leaves, then the runtime will be $log(\frac{n!}{2^n})$.

Trying to basically show a lower bound similar to $n\log n$ that $log(\frac{n!}{2^n})$ supersedes in order to show that it cannot be $O(n)$.

$\rightarrow$ We know that $n! > (\frac{n}{2})^{\frac{n}{2}}$

$\rightarrow \log n! \geq \frac{n}{2}\log(\frac{n}{2}) = \frac{n}{2}(\log n - \log 2) = \frac{n}{2}\log n - \frac{n}{2}\log 2$

$\rightarrow \log(\frac{n!}{2^n} = \log n! - n\log 2 \geq \frac{n}{2}\log n - \frac{n}{2}\log 2 - n\log 2$

$\rightarrow \log(\frac{n!}{2^n}) \geq \frac{n}{2}\log n - \frac{3n}{2}\log 2$

$\rightarrow \frac{n}{2}\log n - \frac{3n}{2} \rightarrow \frac{n}{2}\log n - O(n)$

$\rightarrow \log(\frac{n!}{2^n}) = \Omega(n\log n)$

$\rightarrow \log(\varepsilon * n!) \neq O(n)$

Given that $n\log n$ grows faster than n and $\log\frac{n!}{2^n} \geq n\log n$, meaning that it's either equal to the best case of $n\log n$ (Omega) or that it grows even faster, the runtime can't be equal to $O(n)$.

5. (**Honors Problem**, **, 0 points) $\varepsilon = \frac{2^{\lfloor \frac{n}{2} \rfloor}}{n!}$.

Solution:

Problem 4-2 – Sorting Bounds (20 points)

1. (5 points) Assume you are given an array of n integers in the range $\{1, \ldots, (\log n)^{\log n}\}$. Show how to sort this array in time $O(n \operatorname{loglog} n)$.

Solution: Sort the array using radix sort with a base n. Each number becomes a string of at most $D = O(\log \log n)$ over a size n. For each digit position that we're running radix sort on, we run counting sort on that digit. Counting sort on n items runs in $O(n + n) = O(n)$. Counting sort is already stable, so after processing digits 0 through k, the items are sorted by their low-order $k + 1$ digits.

fxn Sort (A[1...n]):
D = ceiling of (log log n)
for pos = 0 to D - 1:
CountingSortbyDigit(A, pos, base = n) return A where number of digit-positions leads to O(log log n) and each counting sort pass costs O(n), so O(nlog log n).

2. (15 points) Assume you are given an array of n integers with many duplicates, so that you know that there are at most $\frac{n}{\log n}$ distinct elements in the array. Show how to sort this array in time $O(n)$. You may assume access to an empty array of size B where B is an upper bound on the integers in your input as well as any additional memory you may require.

(**Hint:** Adapt counting (small-key) sort by keeping track of which elements exist in the array and how many from each one.)

Solution: Do a single pass in linear time over input array and build frequ table that represents the counts. Also have to create a dynamic list of sorts of the keys that appear, where we're appending the key to this list the first time it appears (so count = 1). This implies that the size of this dynamic list is: $L \leq \frac{n}{\log n}$. Then sort L (which are the unique keys only) with some sort of comparison algo, whose running time is: $\frac{n}{\log n} * \log(\frac{n}{\log n}) = O(n)$. Iterate over L and for each k output count[k] copies, which is still linear time.

```
Input array A, at most n/logn distinct vals
counting and building list of distinct keys and allocating count 0..B
L = empty list
for i = 1..n:
v = A[i]
if count[v] == 0: append v to L
count[v] = count[v] + 1
sorting distinct keys now, using linear comparison
sort(L)
then output in order:
index = 1
for each v in L (sorted alrdy):
for t = 1..count[v]: Asorted[index] = v, index+=1
return Asorted
```

Problem 4-3 – Unstable Radix Sort (10 points)

You are given the array $A = (127,\ 736,\ 989,\ 126)$. We will run radix sort (base 10) for three passes (ones $\rightarrow$ tens $\rightarrow$ hundreds), but suppose we run an **unstable** sorting algorithm in each pass instead of counting sort.

1. (5 points) Fill in the array after each pass:

After ones-digit pass:	126, 736, 127, 989
After tens-digit pass:	127, 126, 736, 989
After hundreds-digit pass:	126, 127, 736, 989

2. (5 points) Briefly explain what went wrong and why instability can (potentially) make the final output incorrect.

Solution: In the array A, instability showed up when sorting by ones-place and tens-place. After the one-digit pass, the two numbers ending in 6: 736 and 126 were reversed even though 736 was first in the original array. Then, during the tens-digits pass, the two numbers with tens-digit 2: 126 and 127 also flipped in order. Even though the reversals didn't end up messing up the final order for this array, the instability allowed this possibility. Radix sorting needs stability to ensure that numbers sharing the same current digit keep relative order from previous passes and the original array. Instability can scramble that ordering.

Problem 4-4 – 3-Way Karatsuba (30 Points)

In this question, we will consider the problem of making 3-way Karatsuba multiplication. In essence, we will divide our numbers X and Y into three parts rather than 2, as so:

$$X = A \cdot 10^{2n/3} + B \cdot 10^{n/3} + C$$

$$Y = D \cdot 10^{2n/3} + E \cdot 10^{n/3} + F.$$

1. (10 points) Describe the naive approach using 9 small-part multiplications to find the product $X \cdot Y$ using the split as above.

Solution: Basically multiply every part of X with every part of Y, which happens to add to 9 small mults. This results in AD, AE, AF, BD, BE, BF, CD, CE, CF. Essentially split the X and Y equations into six separate quantities that can be foiled out. Once we have those products we can assemble them according to how their digit-place shifts:

$$\begin{aligned}
&\to X * Y = AD * 10^{\frac{4n}{3}} \\
&+ (AE + BD) * 10^{n} \\
&+ (AF + BE + CD) * 10^{\frac{2n}{3}} \\
&+ (BF + CE) * 10^{\frac{n}{3}} \\
&+ CF
\end{aligned}$$

2. (10 points) Show how to reduce this number to 6 multiplications using an approach similar to Karatsuba.

Solution: Calculate three base multiplications, starting with diagonal products (of a sense): AD, BE, CF. These should translate to the highest, middle, and lowest digits/positions in base $10^{\frac{n}{3}}$. Then find three cross-combination mults, now with fewer mults:

$$\begin{aligned}
M_1 &= (A + B + C)(D + E + F) \\
M_2 &= (A + B)(D + E) \\
M_3 &= (B + C)(E + F)
\end{aligned}$$

These three products combine 9 base-cross terms. Then expand:

$$\begin{aligned}
M_1 &= AD + AE + AF + BD + BE + BF + CD + CE + CF \\
M_2 &= AD + AE + BD + BE \\
M_3 &= BE + BF + CE + CF
\end{aligned}$$

Just need to re-substitute back in the cross-term expressions from above, all of which are formed from 6 total mults.

3. (10 points) Describe the recurrence relation to find the time complexity of the three-way Karatsuba, and give its asymptotic complexity.

Solution:

$\rightarrow$ split two n-digit numbers into three parts, size roughly $\dfrac{n}{3}$

$\rightarrow X = A10^{\frac{2n}{3}} + B10^{\frac{n}{3}} + C, Y = D10^{\frac{2n}{3}} + E10^{\frac{n}{3}} + F$

$\rightarrow$ 6 mults of size n/3 and O(n) additional work for operations

$\rightarrow T(n) = 6T(n/3) + O(n)$

$\rightarrow O(n^{log_3(6)})$ time complexity

$\rightarrow \Theta(n^{1.6})$ asymptotic complexity with base case T(1)=O(1).

4. (Honors Problem, *, 0 points)) Compare the time complexity derived above to if you were to use 2-way Karatsuba to find the product. Which version is faster?

Solution: