Due September 29 (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, Brilliant, StackOverflow

Problem 3-1 – Super Inversions (10 Points)

Recall that an inversion of an n-long array A is a pair of indices (i, j) such that $1 \leq i < j \leq n$ but $A[i] > A[j]$. In the first homework, you were tasked with creating an algorithm that could find the number of inversions in a given array in $\Theta(n \log n)$ time.

Consider now the task of finding *super inversions*. We define a super inversion as a pair of indices (i, j) such that $1 \leq i < j \leq n$ but $A[i] > 2A[j]$. Give a $\Theta(n \log n)$ algorithm that finds the number of super inversions in a given array A. Briefly justify its runtime.

Solution: I think the idea can be pretty similar to the inversion-algo created in the first homework. We can use a similar divide-and-conquer strategy to count ordinary inversions, where at each merge going up the recurrence tree, we want to make sure that each possible pair satisfies $A]i] > 2A[j]$. When merging the two sorted halves, we would want to count, for each elem in the left half, how many elems in the right half are still less when doubled. Think it's also impt to point out that the j-pointer shouldn't reset for each i, with j only moving forward.

Counting and Sorting Array for Inverions (A, left, right): if left >= right: return 0 mid = (left + right) // 2
count = Counting and Sorting (A, left, mid)
count += Counting and Sorting (A, mid+1, right)
j = mid+1
for i from left to mid: while j <= right and A[i] > 2 * A[j]: j +=1

Then would need to add number of j's in right that satisfy 2* condition for current i
conduct merge, return final count...

Runtime justification:
If what I put down makes sense, each recursion should split array in two (just like in reg merge sort from first homework), creating $\frac{n}{2}$. At each merge, the counting loop iterates over entire array, creating n. This gives a recurrence of $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, resulting in $\Theta(n \log n)$.

Problem 3-2 – Median of Medians (35 Points)

In the linear time "median-of-medians" selection algorithm, we grouped elements into groups of 5 and used the $n/5$ medians of these groups. We showed that the median of the $n/5$ medians has at least $(3/10)n$ elements that are no greater than it, and at least $(3/10)n$ elements that are no smaller than it.

We now ask what would happen if we used groups of 3 or 7 elements instead of 5.

1. (10 points) For a group size of 3, compute the constant $\alpha \in [0, 1]$ that replaces the constant of $3/10$ that we had for groups of 5. In other words, for what value of α is the median of the medians guaranteed to have at least αn elements that are no greater than it, and at least αn elements that are no smaller than it? Explain your answer.

   Solution: Split the array into n/3 groups of 3 and find median for each, which should give n/3 group-medians. Find the median of that set of the medians, meaning that this median has two elems in that group that are $\leq$ than it. This means that the total number of array elems is $\leq$ M: $2[\frac{n}{6}] \leq \frac{n}{3} - \mathcal{O}(1)$, resulting in $\alpha = \frac{1}{3}$, $\frac{1}{3}n$ elems no greater than main median and at least $\frac{1}{3}n$ elems no smaller than it for groups of 3.

2. (5 points) Again find the constant α, now for group size of 7. Explain your answer.

   Solution: Similarly sploitting elements now into n/7 groups and finding main median. There are at least $(\frac{n}{7}/2 = \frac{n}{14}$ groups that have medians less than the main median. If the group median is $\leq$ the main median, at least four smallest elemes in the group are also $\leq$ the group median. This results in $4 * \frac{n}{14} = \frac{4}{14}n - \mathcal{O}(1) = \frac{2}{7}n -$ constant time. So $\alpha = \frac{2}{7}$.

3. (10 points) Compute the running time of the median-of-medians selection algorithm if we replace the group size by 3. Write the recurrence for the running time and use the Recursion Tree method to solve it.

   Solution: If we replace with group size 3, we split into n/3 groups and find each median in linear time. We recursively select the median of those medians in n/3 time and partition arnd that pivot in linear. We then recurse into the larger half, which is at most (2/3)n. The recurrence is: $T(n) \leq T(\frac{n}{3}) + T(\frac{2n}{3}) + cn$.

Recursion Tree method:
Work being done at root level is n.
As tree branches down into n/3 groups, the size will only add up to the total elems, meaning total work at any level is still c * n.
A child will be (at its largest) 2/3 of its parent. So, $(\frac{2}{3})^t n = \mathcal{O}(1) \rightarrow t = \mathcal{O}(\log n)$ work being done, meaning tree has O(logn) levels. Multiplying this by the linear time on each level results in $\Theta(n \log n)$.

4. (10 points) Now compute the running time of the median-of-medians selection algorithm if we replace the group size by 7. Write the recurrence for the running time and use the Recursion Tree method to solve it.

Solution: We instead form n/7 groups and find each median in linear time, select median of medians in n/7 time, and partition around that value in linear. Recursing on larger half results in $(1-a)n = (1-\frac{2}{7})n = \frac{5n}{7}$, resulting in recurrence $T(n) \leq T(\frac{n}{7}) + T(\frac{5n}{7}) + cn$.

Recursion Tree method:
At the root level before it branches, there's linear work being done
The next level will have two childs of size n/7 and 5n/7, so total work being done is still linear (with some constant attached)
Each child spawns two more childs (at most), who are at most the 6/7 the size of the prev parent. This should also result in linear work apart from the 6/7 constant.
Summing all the work being done results in the same as the levels: $\Theta(n)$.

Problem 3-3 – Mushroom Majority (25 Points)

You are an assistant at a fancy restaurant. A farmer has come to you with a large basket of n truffles (a fancy type of mushroom). The truffles are of an unknown number of different varieties. The head chef wants to make a sauce with at least $n/2 + 1$ truffles of *the same variety* and will only buy the basket if he can make a suitable sauce. You cannot tell the difference between the varieties, but thankfully the farmer has brought along his truffle-sniffing pig. You can give the pig any two mushrooms, and it will oink if and only if two truffles held to its snout are of the same variety.

1. (10 points) Show how to decide whether to buy the basket with $O(n \log n)$ queries to the pig.

Solution: So it's impt to keep in mind that all we have is something to verify whether two elems are equal (pig). The idea is that we want to maybe iterate once through the array of truffles and make pairs out of each pair in the array (tuple). If the pig says that the pair is diff, discard both truffles and if they're the same, just keep one as a representative of the pair. The algo should repeat the steps to find pairs until the array has discarded everything except one truffle or 0. If the algo results with one truffle remaining, do another iteration through the original to sum how many truffles the winner is equal to. It should work because every time an unmatching pair is discarded we're still maintaining the idea that there's variety > n/2 times.

2. (10 points) Justify your runtime.

> **Solution:** If T(n) is the number of pig-queries, we should be making two recursive calls, which are m/2 work done for each. The surviving pairs/indiv truffles are at most n/2 in count, so size will half after each pass through. Total number of pig queries across all query rounds through array is at most $[\frac{n}{2}] + [\frac{n}{4}] + [\frac{n}{8}] + \ldots$. Final verification with one winner does another query thru entire array in linear time. With the final linear pass and indiv comparisons as the size halves, the runtime should result in $\mathcal{O}(n \log n)$.

3. (5 points) Modify your solution to also collect the $n/2 + 1$ mushrooms of the same variety (if this grouping exists).

> **Solution:** Do the same pair-discard through the array to finalize a winner that has matches. Then do the same verification that's being done in the prev. solution but count and collect mushrooms until it matches n/2 + 1. The main modifications are mainly in what is being kept and how the final verification happens. We want to store an index for the final winner and then actually store the indices/elems during the final verif. so we can output. The verification would also collect matches rather than just count thru the array like in the prev soln. The verif will stop early once n/2 + 1 items are collected.

4. (honors problem, 0 points, **) Provide an algorithm that decides on the basket in only $O(n)$ pig queries. (hint: If you discard pairs of truffles from different varieties, what will happen if there is indeed a majority variety?)

Solution:

Problem 3-4 – Repeat Elements (30 Points)

Consider an array A with n elements where it is guaranteed that every element appears exactly twice in A, e.g., $A = (9, 7, 7, 1, 9, 1, 3, 5, 3, 5)$. For any two elements $A[i], A[j]$ in the array, we may only compare the elements by testing equality, i.e., $A[i] \stackrel{?}{=} A[j]$. With this in mind,

1. (15 points) Give an algorithm that returns two positions in A that have the same element using at most $n - 2$ comparisons/equality tests.

   Note: These may be any two positions, so for example $(1, 5), (2, 3), (4, 6), (7, 9)$ or $(8, 10)$ are all valid outputs of this algorithm on A.

Solution: Pick a fixed element such as elem at index 0 and compare to every other element except the last. If there's a match, return, otherwise the pair has to be the last one with the first. Every value will appear exactly twice, which uses at most $n - 2$ comparison tests.

for j = 2 to n-1:
if A[1] == A[j]:
return 1,j
If it reaches the end, then that means A[1] doesn't equal A[j] for loop condition, implying that only match is for first elem is the last.

2. (15 points) Prove that your algorithm really needs $n - 2$ comparisons in the worst case (i.e., there are inputs where it uses that many comparisons before terminating). Specifically, give an example of a worst-case input for $n = 10$.

Solution:

$\rightarrow n = 10$, 0-based
$\rightarrow A = 1, 7, 7, 3, 3, 5, 5, 4, 4, 1$
$\rightarrow$ Pairs include: $(0, 9), (1, 2), (3, 4), (5, 6), (7, 8)$

This is the worst-case because the algo needs to compare the first elem with each elem in the array before reaching the last one by default, which is the only one that matches. The loop performs the full 8 comparisons before returning.

Honors Problem – Lower Bounds (0 Points)

In Question 3-4, we asked you to give an algorithm to find two positions with the same element.

- (****) We conjecture that any correct comparison-based deterministic algorithm must use *at least* $n - 2$ comparisons. Is the conjecture true? Can you prove it?

Solution:

- (**) However, it is possible to construct a *randomized* algorithm which in expectation uses fewer comparisons. Suggest a randomized algorithm and state (try to justify) how many comparisons it requires in expectation. (Hint: suppose you just pick a few positions of the array at random. How many do you need to pick before there is a good chance that some element occurs at two of those positions?)

Solution: