Solution 2: Searching and Sorting

Problem 1. Let $X = [a_1, \cdots, a_n]$ and $Y = [y_1, \cdots, y_n]$ be two sorted arrays (in non-decreasing order). For simplicity, assume n is a power of 2.

(a) Describe an algorithm to find the median of all $2n$ elements in the arrays X and Y in $O(\log n)$ time.

(b) Provide a succinct proof of the correctness of the algorithm.

(c) Provide an analysis of the running time (asymptotic analysis is correct) and memory utilization of the algorithm.

Hint: Note that the given arrays are already sorted and of the **same size**! You may want to use binary search to exploit this fact. :)

Solution. 1. We use the following algorithm:

- If $n = 1$ or $n = 2$ then merge the arrays X and Y manually and find the median of the new array.
- If $n > 2$, then we'll first compute $X_{mid} = \lfloor \frac{n}{2} \rfloor$ and $Y_{mid} = n - 1 - X_{mid}$.
  (a) (condition 1) If $(X[X_{mid}] \leq Y[Y_{mid}+1])$ and $(Y[Y_{mid}] \leq X[X_{mid}+1])$, then identify the 2 largest numbers from the list $[Y[Y_{mid}-1], Y[Y_{mid}], X[X_{mid}], X[X_{mid}-1]$ and return their average.
  (b) (condition 2) If $X[X_{mid}] > Y[Y_{mid}+1]$, then recur on $X[0 : X_{mid}]$ and $Y[Y_{mid}+1 :]$, and return its result.
  (c) (condition 3) Otherwise (ie. $Y[Y_{mid}] > X[X_{mid}+1]$), then recur on $X[X_{mid}+1]$ and $Y[0 : Y_{mid}]$, and return its result.

2. We proceed with strong induction on the number of elements k. For $k \leq 2$, we immediately get the result as we compute the median directly. Now, suppose that for $k = \{3, \ldots, n\}$, $P(k)$ holds. We'll prove that $P(n+1)$ holds.

   If condition 1 holds, all of the elements in $X[0 : X_{mid}+1] \bigcup Y[0 : Y_{mid}+1]$ must be less than or equal to all of the elements from $X[X_{mid}+1 :] \bigcup Y[Y_{mid}+1 :]$. Then, the median will simply be the average of the two largest numbers from $X[0 : X_{mid}+1] \bigcup Y[0 : Y_{mid}+1]$.

   If condition 2 holds, X_{mid} should be at a lower index in X (inversely, Y_{mid} should be at a greater index in Y). The maximum length of the sub-arrays we recur on are at least of length 1 and at most length n. Recurring on these inputs should yield the correct solution.

   If condition 3 holds, X_{mid} should be at a greater index in X. As previously explained, recurring on the new sub-arrays should yield the correct solution.

   Having exhausted all the cases, by strong induction we can conclude the algorithm's correctness holds for all k, as desired.

3. **Time Complexity**: At each step of the algorithm, we halve the length of the input arrays and perform a fixed series of constant time computations. Thus, the runtime of our algorithm is $O(\log(n))$.

   In practice, our approach uses a left and right pointer to keep track of the input sub-arrays. We also temporarily allocate space for X_{mid} and Y_{mid}, so our overall memory complexity is $O(1)$.

□

Problem 2. Let A be an array of n *distinct* integers. An *inversion* in A is a pair of indices i and j such that $i < j$, but $A_i > A_j$. For example, the following sequence has three *inversions*:

$$\{1,\ 5,\ 2,\ 8,\ 4\}$$
$$(5,2),\ (5,4),\ (8,4)$$

(a) Provide a succinct (but clear) description of an algorithm running in $O(n \log n)$ time to determine the number of *inversions* in A. You may provide a pseudocode.

(b) Provide a succinct proof of the correctness of the algorithm.

(c) Provide an analysis of the running time (asymptotic analysis is correct) and memory utilization of the algorithm.

Solution. (a) We use the following modification of MergeSort:

Algorithm 1 Counting Inversions

```
Input: Two sorted arrays to merge, leftArr and rightArr
Output: A pair containing the merged sorted array and the number of inversions
 1: function ModifiedMerge(leftArr, rightArr)
 2:     outputArr ← []
 3:     inversionCounter ← 0
 4:
 5:     while leftArr and rightArr are not empty do
 6:         if rightArr is empty or leftArr[0] ≤ rightArr[0] then
 7:             remove the first element of leftArr and append it to outputArr
 8:         else if leftArr is empty or rightArr[0] < leftArr[0] then
 9:             increment inversionCounter by length of left Arr
10:             remove the first element of rightArr and append it to outputArr
11:         end if
12:     end while
13:     return (outputArr, inversionCounter )
14: end function

Input: An array to sort and optionally the total number of inversions so far
Output: The sorted array and the number of inversions required to sort the array
15: function ModifiedMergeSort(arr, numberOfInversions = 0)
16:     if the arr has length ≤ 1 then
17:         return (arr, 0)
18:     end if
19:
20:     leftArr, leftInversions ← ModifiedMergeSort(left half of arr)
21:     rightArr, rightInversions ← ModifiedMergeSort(right half of arr)
22:     sortedArr, mergeInversions ← ModifiedMerge(leftArr, rightArr)
23:     return (sortedArr, leftInversions + rightInversions + mergeInversions )
24: end function
```

(b) This algorithm correctly counts the number of *inversions* in an input array. The order of the input numbers in MergeSort are changed during the merge step. Since the arrays to ModifiedMerge are sorted, if the leftmost element of the right array is smaller than the leftmost element of the left array, we know that the leftmost element of the right array is smaller than all of the elements in the left array. Therefore, we only increment our counter every time we encounter an inversion.

To see that every inversion is counted, note that for an inversion between two elements a, b, the mergesort algorithm will always split them apart, so there is always a step at which we join two arrays where one contains a and the other contains b. This ensures that this inversion is always counted (and never double counted!), meaning we return exactly the number of inversions in the entire array.

(c) The modifications required to MergeSort to count inversions does not affect the overall big-O runtime class as it is constant time comparison. Incrementing a counter in ModifiedMerge will affect the runtime of ModifiedMerge by a constant. Therefore, the overall runtime of ModifiedMergeSort is in the same asymptotic class as MergeSort which runs in $O(n \log n)$ time.

□

Problem 3. Let $A_1, \cdots, A_k$ be k arrays where each $A_i = [A_{i1}, \cdots, A_{in}]$ is sorted in ascending order. For simplicity, assume that all arrays have the same length n and the total number of elements across all arrays is $N = k \times n$.

(a) Describe an algorithm to to merge all k sorted arrays into a single sorted array in $O(N \log k)$ time.

(b) Provide a succinct proof of the correctness of the algorithm.

(c) Provide an analysis of the running time (asymptotic analysis is correct) and memory utilization of the algorithm.

Solution. (a) Consider the following modification of mergesort:

Algorithm 2 Merging k Sorted Arrays

```
Input: A list of k sorted arrays, arrays
Output: A single merged sorted array
 1: function MergeTwoArrays(A, B)
 2:     result ← []
 3:     while array A and array B are not empty do
 4:         if A[0] < B[0] then
 5:             remove the first element of array A and append it to outputArr
 6:         else
 7:             remove the first element of array B and append it to outputArr
 8:         end if
 9:     end while
10:     return output
11: end function

Input: A list of k sorted arrays, arrays
Output: A single merged sorted array
12: function MergeKSortedArrays(arrays)
13:     if length of arrays ≤1 then
14:         return arrays[0]
15:     end if
16:     mid ← length of arrays // 2
17:     leftHalf ← MergeKSortedArrays(left half of arrays list)
18:     rightHalf ← MergeKSortedArrays(right half of arrays list)
19:     return MergeTwoArrays(leftHalf, rightHalf)
20: end function
```

This algorithm effectively merges k sorted arrays into a single sorted array by leveraging a method similar to merge sort. Initially, the merging process is applied to two arrays at a time. The mergeTwoArrays function efficiently combines two sorted arrays into one by comparing the smallest unmerged elements from each array and appending the smaller element to the

result. This process continues until all elements from both arrays are included in the result. In the mergeKSortedArrays function, this merging strategy is recursively applied to progressively larger groups of arrays. By dividing the array of k arrays into two halves and recursively merging these halves, the algorithm ensures that at each step, sorted arrays are combined into larger sorted arrays.

(b) We provide a proof using induction on the number of arrays. In our base case, $k = 1$, the input is a single sorted array. The algorithm MergeKSortedArrays returns this array directly since there is no need for merging. This is correct by definition, as a single sorted array is trivially sorted.

For the inductive hypothesis, assume that the algorithm MergeKSortedArrays works correctly for at most K sorted arrays. That is, the algorithm merges k sorted arrays into a single sorted array correctly for $k \leq K$.

Now consider sorting $K + 1$ arrays. MergeKSortedArrays will split k+1 arrays into two halves of $\lceil \frac{k+1}{2} \rceil$ arrays. When recursively applying MergeKSortedArrays to each half, by the induction hypothesis, each recursive call correctly merges the sorted arrays in its half into a single sorted array. Then, we combine the two arrays using the same algorithm as in the combine step of mergesort, which we already know to be correct. By strong induction, the algorithm's correctness holds for all k.

(c) **Time Complexity**: Since each merge operation (in MergeTwoArrays) takes linear time in the number of elements being merged, and the merging is performed at each level of recursion, the overall time complexity of merging k arrays is $O(N \log k)$, where N is the total number of elements across all arrays.

The memory utilization is composed of memory for utilizing input array ($O(N)$), intermediate arrays during the merging process which is $O(N)$ across all levels of recursion and memory for the recursive stack itself, $O(\log k)$. Therefore, this algorithm uses a linear amount of memory. □