Problem 1. (Faster Algorithm for Finding the Number of Crossing Inversions) Let S_1 and S_2 be two disjoint sets of n integers. Assume that S_1 is stored in an array A_1, and S_2 in an array A_2. Both A_1 and A_2 are sorted in ascending order. Design an algorithm to find the number of such pairs (a, b) satisfying all of the following conditions: (i) $a \in S_1$, (ii) $b \in S_2$, and (iii) $a > b$. Your algorithm must finish in $O(n)$ time.

Solution. Iterate through array A_1 and for each $A_1[i]$ count the number of elements in A_2 smaller than $A_1[i]$. Since A_2 is non-decreasing, we can find the largest j such that $A_2[j] < A_1[i]$ by scanning A_2 once, and add to the number of inversions by j. This gives rise to a $O(n^2)$ algorithm.

To save the number of comparisons wasted on scanning A_2, we maintain a pointer j which points to A_2, and shift j accordingly before counting the number of inversions involving $A_1[i]$. Observe that A_1 is non-decreasing, this means that $A_2[k] < A_1[i] < A_1[i + 1]$ for every $k \leq j$. Therefore, j never decreases. Noting that we only iterate through arrays A_1, A_2 once, this streamlined algorithm should run in $O(n)$ time.

```
procedure CrossingInversions($A_1, A_2$)
    $j \leftarrow 0, cnt \leftarrow 0$
    for $i \leftarrow 1$ to n do
        Increase $j$ by 1 until $j + 1 > n$ or $A_2[j + 1] \geq A_1[i]$
        $cnt \leftarrow cnt + j$
    end for
    return $cnt$
end procedure
```

Problem 2. (Faster Algorithm for Finding the Number of Inversions) Given an array A of n integers, design an algorithm to find the number of inversions in $O(n \log n)$ time.

Solution. Besides reporting the number of inversions, we also sort the array in ascending order.

Divide: Let A_1 be the array containing the first $\lceil n/2 \rceil$ elements of A, and A_2 be the array containing the other elements of A. Then, recurse on A_1, A_2 to find the number of inversions in A_1, A_2, respectively.

Conquer: Apply cross inversion counting on the two sorted arrays. Then, merge the two arrays. Both operations can be done in $O(n)$ time.

We obtain the recursive formula: $f(n) = 2f(n/2)+O(n)$, which then resolves into $O(n \log n)$ time.

Problem 3. (General Matrix Multiplication) In the class, we explained how to multiply two $n \times n$ matrices in $O(n^{2.81})$ time when n is a power of 2. Explain how to ensure the running time for any value of n.

Solution. Let m be the smallest power of 2 that is larger than n. Obtain an $m \times m$ matrix A' by padding $m - n$ dummy rows and columns to A with only 0 in each entry, and similarly, an $m \times m$ matrix B' from B. Compute $A'B'$ by Strassen's Algorithm in $O(m^{2.81}) = O((2n)^{2.81}) = O(n^{2.81})$ time. Then, the matrix AB can be obtained by discarding the last $m - n$ rows and columns from the matrix $A'B'$.

Problem 4. (Faster Dominance Counting) Give an algorithm of $O(n \log n)$ expected time to solve the dominance counting problem.

Solution. We first sort the n points by their x-coordinates in ascending order, in $O(n \log n)$ time.

Then, we apply divide-and-conquer, besides counting the number of dominance count for each point, we sort the array by y-coordinates in ascending order.

Divide: Separate the points with a vertical line ℓ such that at most $\lceil n/2 \rceil$ points are on each side of the line. Let A_1 be the array containing the points on the left of the vertical line, and A_2 containing the rest of the points. Recurse on A_1, A_2 to find for each point $p_1 \in A_1$ the number of points in A_1 that dominates p_1, and for each $p_2 \in A_2$ the number of points in A_2 that dominates p_2. Besides, recurse to sort A_1, A_2 by y-coordinate, in ascending order.

Conquer: We know that no points in A_1 can dominate A_2. Therefore, we only need to count for each $p_2 \in A_2$, the number of points in A_1 dominated by p_2. Now that A_1, A_2 are in ascending order of y-coordinate, we can report the number of points in A_1 having their y-coordinate smaller than or equal to that of p_2, for each point $p_2 \in A_2$, in $O(n)$ time by monotonicity. We also spend $O(n)$ time to merge the two ordered list, to ensure that the y-coordinate of the points of $A_1 \cup A_2$ are in non-decreasing order.

We obtain the recursive formula: $f(n) = 2f(n/2) + O(n)$, which then resolves into $O(n \log n)$ time.

Problem 5. (Subarray Sum Problem) Let A be an array of n integers (A is not necessarily sorted). Each integer in A may be positive or negative. Given i, j satisfying $1 \le i \le j \le n$, define sub-array $A[i : j]$ as the sequence $(A[i], A[i+1], \ldots, A[j])$, and the weight of $A[i : j]$ as $A[i] + A[i+1] + \ldots + A[j]$. For example, consider $A = (13, -3, -25, 20, -3, -16, -23, 18)$; $A[1 : 4]$ has weight 5, while $A[2 : 4]$ has weight -8.

(a) Give an algorithm to find a sub-array of with the largest weight, among all sub-arrays $A[i : j]$ with $j = n$. Your algorithm must finish in $O(n)$ time.

(b) Give an algorithm to find a sub-array with the largest weight in $O(n \log n)$ time (among all the possible sub-arrays).

Solution.

(a) Suffix sum. We create an auxiliary array storing the sum of the each subarrays $A[i : n]$ ($1 \le i \le n$), and then we can find the maximum weight among the n values in $O(n)$ time. The creation of the auxiliary array can be done in $O(n)$ time: scan A from $A[n]$ to $A[1]$ and maintain the sum s of the elements already being scanned. When we encounter $A[i]$, first increment s by $A[i]$; next, store s to the auxiliary array indicating the weight of the subarray $A[i : n]$.

(b) Apply divide-and-conquer. **Divide**: Split the array into two halves: $A_1 := A[1 : \lceil n/2 \rceil]$ and $A_2 := A[\lceil n/2 \rceil + 1, n]$. Recurse to find the subarray in A_1, A_2 with maximum weight, respectively. **Conquer**: We try to test whether the subarrays straddling across A_1 and A_2 could be candidates maximizing the subarray sum. Create two auxiliary arrays storing the suffix sum of A_1 and the prefix sum of A_2. Then, pick in each of the array the maximum element, indicating the sum of $A[i_1, \lceil n/2 \rceil]$ and the sum of $A[\lceil n/2 \rceil + 1, i_2]$. The sum of these two maximum element indicates the maximum possible weight of a subarray $A[i_1, i_2]$, straddling across A_1, A_2. Taking the maximum among the aforementioned, the maximum subarray sum contained in A_1, A_2, finishes the conquering step.

Problem 6. Given an array A of size n, design an algorithm to output all the inversions in A using $O(n \log^2 n + k)$ time, where k is the number of inversions reported.

Solution. Apply divide-and-conquer.

Divide: Let A_1 be the array containing the first $\lceil n/2 \rceil$ elements of A, and A_2 be the array containing the other elements of A. Solve the problem of "inversion reporting" recursively on A_1 and A_2, respectively.

Conquer: It remains to report the cross inversion pairs (i, j) where $i \in A_1$ and $j \in A_2$. First, sort A_1 in $O(n \log n)$ time. Then, for each element $A_2[j]$ we can find the first $A_1[i]$ satisfying $A_1[i] > A_2[j]$ by binary search. We report all pairs $(A_1[k], A_2[j])$ with $k \geq i$. This takes $O(n \log n + n) = O(n \log n)$ time in total.

We obtain the recursive formula: $f(n) = 2f(n/2) + O(n \log n)$, which then resolves into $O(n \log^2 n)$ time.

Problem 7. Prove: if you can solve the dominance counting on n points in $f(n)$ time, then you can count the number of inversions in an integer array of length n in $f(n) + O(n)$ time. (Hint: you can convert the inversion counting problem to an instance of dominance counting.)

Proof. Create a set of points in $O(n)$ time corresponding to each element $A[i]$ in A with coordinate $(-i, A[i])$. Notice that $(A[i], A[j])$ forms an inversion pair when $i < j$ and $A[i] > A[j]$ if and only if the corresponding points $(-i, A[i])$ dominates $(-j, A[j])$. Therefore, the sum of the dominance count of each point is equal to the number of inversions. This then follows that any algorithm that solves dominance counting in $f(n)$ time solves the inversion counting problem in $f(n) + O(n)$ time. $\square$

Problem 8. Assuming $m \geq n$, give an algorithm to multiply an $m \times n$ matrix with an $n \times m$ matrix in $O(m^2 \cdot n^{0.81})$ time. (Hint: apply Strassen's algorithm to multiply $\lceil m/n \rceil^2$ pairs of order-n matrices.)

Solution. Let m' be a multiple of n that is greater than or equal to m. Obtain a $m' \times n$ matrix A by padding $m' - m$ dummy rows with 0 in every entry in $O(mn)$ time. Similarly, obtain an $n \times m'$ matrix B by padding $m' - m$ dummy columns with 0 in every entry. Partition A and B into block form such that A contains m'/n rows of $n \times n$ blocks and B contains m'/n columns of $n \times n$ blocks. This can also be done in $O(mn)$ time.

Obtain AB by applying Strassen's algorithm. In particular, we multiply these $(m'/n)^2$ pairs of $n \times n$ matrices in $O((m'/n)^2 \cdot n^{2.81}) = O((2m/n)^2 \cdot n^{2.81}) = O(m^2 \cdot n^{0.81})$ time. The result of multiplying the given $m \times n$ and $n \times m$ matrices can be obtained by discarding the last $m' - m$ rows and the last $m' - m$ columns of AB. Thus, the total time complexity is $O(mn + m^2 \cdot n^{0.81}) = O(m^2 + m^2 \cdot n^{0.81}) = O(m^2 \cdot n^{0.81})$.

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_{m'/n} \end{bmatrix}, \quad B = \begin{bmatrix} B_1 & B_2 & \dots & B_{m'/n} \end{bmatrix}, AB = \begin{bmatrix} A_1B_1 & A_1B_2 & \dots & A_1B_{m'/n} \\ A_2B_1 & A_2B_2 & \dots & A_2B_{m'/n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m'/n}B_1 & A_{m'/n}B_2 & \dots & A_{m'/n}B_{m'/n} \end{bmatrix}$$

Problem 9. Assuming $m \geq n \geq t$, give an algorithm to multiply an $m \times n$ matrix with an $n \times t$ matrix in $O(m \cdot n \cdot t^{0.81})$ time. (Hint: apply Strassen's algorithm to multiply pairs of $t \times t$ matrices.)

Solution. Let m' be a multiple of t that is greater than or equal to m. Similarly, let n' be a multiple of t that is greater than or equal to n. Obtain a $m' \times n'$ matrix A by first padding $m' - m$ dummy rows with 0 in every entry and then padding $n' - n$ dummy columns with 0 in every entry in $O(mn)$ time. Similarly, obtain an $n' \times t$ matrix B by padding $n' - n$ dummy rows with 0 in every entry in $O(nt)$ time. Partition A and B into block form such that A contains m'/t rows and n'/t columns of $t \times t$ blocks and B contains n'/t rows of $t \times t$ blocks. This can also be done in $O(mn + nt)$ time.

Obtain AB by applying Strassen's algorithm. In particular, we multiply these $(m'/t) \times (n'/t)$ pairs of $t \times t$ matrices in $O((m'/t) \cdot (n'/t) \cdot t^{2.81}) = O((2m/t) \cdot (2n/t) \cdot t^{2.81}) = O(m \cdot n \cdot t^{0.81})$ time, then we sum up these $t \times t$ matrices for the m'/t row-blocks of AB in $O((m'/t) \cdot (n'/t) \cdot t^2) = O(mn)$ time. The result of multiplying the given $m \times n$ and $n \times t$ matrices can be obtained by discarding the last $m' - m$ rows of AB. Thus, the total time complexity is $O(m \cdot n \cdot t^{0.81})$.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,n'/t} \\ A_{2,1} & A_{2,2} & \dots & A_{2,n'/t} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m'/t,1} & A_{m'/t,2} & \dots & A_{m'/t,n'/t} \end{bmatrix}, \ B = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_{n'/t} \end{bmatrix}$$

$$AB = \begin{bmatrix} A_{1,1}B_1 + A_{1,2}B_2 + \ldots + A_{1,n'/t}B_{n'/t} \\ A_{2,1}B_1 + A_{2,2}B_2 + \ldots + A_{2,n'/t}B_{n'/t} \\ \vdots \\ A_{m'/t,1}B_1 + A_{m'/t,2}B_2 + \ldots + A_{m'/t,n'/t}B_{n'/t} \end{bmatrix}$$

Problem 10. Let $A_1, A_2, \ldots, A_k$ be k arrays, each of which has been sorted. These arrays are mutually disjoint, namely, no integer can appear in more than one array. Design an algorithm to merge the k arrays into one sorted array in $O(n \log k)$ time, where n is the total length of the k arrays. Note: these arrays may have different lengths.

Solution. Assume that the length of array A_i is n_i, and each array is sorted in ascending order (otherwise we can reverse the array in $O(n)$ time). Create k pointers pointing to an element of each array that has not been added to the merged list. Initially, these pointers should point to the first element of their corresponding arrays. Furthermore, create a heap with size at most k, which is capable of (i) inserting an element; and (ii) popping the smallest element in the heap. Now, we add the first element of each array into the heap, and shift all the pointers by one element.

In each iteration, we do the following operations:

- Pop the smallest element in the heap and append it to the merged list.
- Let the element just popped originally be in array A_i. We insert the element being pointed by the A_i's designated pointer to the heap, and shift this pointer by one element which is immediately after the one being inserted.

We note that: (i) the heap has at most k elements during the run of the algorithm and thus each operation on the heap costs at most $O(\log k)$ time; (ii) all the elements will be inserted to the heap and popped afterwards, but for only once. We then conclude that the algorithm runs in $O(n \log k)$ time.