CSCI3160: Regular Exercise Set 2

Prepared by Yufei Tao

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 (we gave an $O(n \log n)$-time algorithm in the class).

Solution. Merge $A_1$ and $A_2$ into one sorted list $A$, which takes $O(n)$ time. Scan the elements of $A$ in ascending order. In the meantime, maintain the number $t$ of elements that (i) originate from $A_2$, and (ii) have already been scanned so far: this can be done by setting $t$ to 0 in the beginning, and incrementing it each time an element originating from $A_2$ is scanned. Furthermore, also maintain a counter $c$ as follows: $c = 0$ in the beginning; every time an element originating from $A_1$ is seen, increase $c$ by the current value of $t$. The final $c$ at the end of the algorithm is the number of crossing inversions.

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. We will solve a more challenging problem: besides reporting the number of inversions, the algorithm also needs to sort $A$ in ascending order. Break $A$ at the middle into two arrays $A_1$ and $A_2$ each with at most $\lceil n/2 \rceil$ elements. Recursively, find the number $c_1$ of inversions in $A_1$ and the number $c_2$ of inversions in $A_2$. At this moment, both $A_1$ and $A_2$ have been sorted. We can then apply the algorithm in Problem 1 to find the number of crossing inversions in $O(n)$ time. Finally, merge $A_1$ and $A_2$ into a sorted array using $O(n)$ time. It is rudimentary to verify that the running time is $O(n \log n)$.

Problem 3. Give an $O(n \log n)$-time algorithm to solve the dominance counting problem discussed in the class.

Solution. We will solve a more challenging problem: besides reporting the dominance counts, the algorithm should also sort $P$ in ascending order.

As discussed in the class, our original algorithm divides $P$ into two halves $P_1$ and $P_2$ using a vertical line $\ell$, and then recurse on $P_1$ and $P_2$ respectively. Upon returning from the recursion, the points of $P_1$ and $P_2$ have been sorted by $y$-coordinate. We still need to find, for each point $p_2 \in P_2$, the number of points $p_1 \in P_1$ that are dominated by $p_2$. Next we show that this can be done in $O(n)$ time. Merge $P_1$ and $P_2$ into one sorted list $P$, where the points are sorted in ascending order by $y$-coordinate. Scan $P$. In the meantime, maintain the number $t$ of points that (i) originate from $P_1$, and (ii) have already been scanned so far. Every time a point $p_2$ originating from $P_2$ is seen, the number of points $p_1 \in P_1$ dominated by $p_2$ is precisely the current value of $t$. To complete the algorithm, return the sorted list of $P$. The overall time complexity now becomes $O(n \log n)$.

Problem 4 (Section 4.1 of the Textbook). 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 $1 \leq i \leq j \leq n$, define sub-array $A[i : j]$ as the sequence $(A[i], A[i + 1], ..., A[j])$, and the weight of $A[i : j]$ as

1. 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.

2. 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. Subproblem 1: Scan the elements of A from A[n] to A[1]. At any time, maintain the sum s of the elements already scanned: at the beginning s = 0; after scanning an element A[i], add A[i] to s. Every time we finish doing so for element A[i], the current value s is precisely the weight of A[i : n]. In this way, we obtain the weights of all sub-arrays A[n : n], A[n − 1 : n], ..., A[1 : n] (in this order) in O(n) time. The maximum weight can then be found easily.

Subproblem 2: Break A into two halves: array A1 which contains the first ⌈n/2⌉ elements, and array A2 which contains the rest. Recursively, find the sub-array of A1 with the largest weight, and then the sub-array of A2 with the largest weight. It remains to consider the “crossing” sub-arrays A[i : j] where i ≤ ⌈n/2⌉ and j > ⌈n/2⌉. In particular, we want to find the “best” crossing sub-array, i.e., the one with the maximum weight. Then, the sub-array to output can be decided easily from the three sub-arrays aforementioned.

We say that a sub-array A1[i : j] is right grounded if j = ⌈n/2⌉, and a sub-array A2[i : j] is left grounded if i = 1. A crucial observation is that the “best” crossing sub-array must be the concatenation of

• the right grounded sub-array in A1 with the maximum weight, and
• the left grounded sub-array in A2 with the maximum weight.

From Subproblem 1, we know that each of the above two grounded sub-arrays can be found in O(n) time.

Therefore, if f(n) is the time of solving the problem on an array of length n, it holds that f(n) ≤ 2 · f(⌈n/2⌉) + O(n), which gives f(n) = O(n log n).

Problem 5. In the class, we explained how to multiply two n × 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. If n is not a power of 2, let m be the smallest power of 2 that is larger than n. If A, B are the n × n input matrices, obtain an m × m matrix A’ by padding m − n dummy rows and columns to A containing only 0 values, and similarly, an m × m matrix B’ from B. Calculate A’B’ 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’.