CIS606 Analysis of Algorithms

Fall Semester, 2023

The Divide and Conquer Technique

1. Solve the following recurrences (you may use any of the methods we studied in class). Make your bounds as small as possible (in the big-O notation). For each recurrence, $T(n)$ is constant for $n \leq 2$.

   (a) $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$.

   Answer: For this problem, none of the cases of Master theorem can apply. By expanding the recurrence, we have

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n \log n \\
&= 4T(\frac{n}{4}) + n \log \frac{n}{2} + n \log n \\
&= 8T(\frac{n}{8}) + n \log \frac{n}{4} + n \log \frac{n}{2} + n \log n \\
&= \cdots \\
&= 2^i \cdot T(\frac{n}{2^i}) + n \log \frac{n}{2^{i-1}} + \cdots + n \log \frac{n}{4} + n \log \frac{n}{2} + n \log n
\end{aligned}
$$

   The expanding will not stop until $\frac{n}{2^k} = 1$, that is, $k = \log n$. Hence, we have the following:

$$T(n) = 2^k \cdot T(\tfrac{n}{2^k}) + n \log \tfrac{n}{2^{k-1}} + \cdots + n \log \tfrac{n}{4} + n \log \tfrac{n}{2} + n \log n$$

   Since $2^k = n$ and $T(1) = O(1)$, $2^k \cdot T(\frac{n}{2^k}) = n \cdot O(1)$. Note that $\log \frac{n}{2^{k-1}} = \log n - (k-1)$. We can deduce the following

$$
\begin{aligned}
& n \log \frac{n}{2^{k-1}} + \cdots + n \log \frac{n}{4} + n \log \frac{n}{2} + n \log n \\
&= n(\log n - (k-1)) + \cdots + n(\log n - 2) + n(\log n - 1) + n \log n \\
&= k \cdot n \log n - n[(k-1) + \cdots + 2 + 1] \\
&= kn \log n - n \cdot \frac{k(k-1)}{2} \\
&= kn(\log n - \frac{k-1}{2})
\end{aligned}
$$

   Since $k = \log n$, we have $\log n - \frac{k-1}{2} = (\log n + 1)/2$.
   Therefore, we obtain $T(n) = n \cdot O(1) + n \log n \cdot \frac{\log n + 1}{2} = O(n \log^2 n)$.

2. Let $A[1 \ldots n]$ be an array of n elements and $B[1 \ldots m]$ another array of m elements, with $m \leq n$. Note that neither A nor B is sorted. The problem is to compute the number of elements of A that are smaller than $B[i]$ for each element $B[i]$ with $1 \leq i \leq m$. For simplicity, we assume that no two elements of A are equal and no two elements of B are equal.

   For example, let A be $\{30, 20, 100, 60, 90, 10, 40, 50, 80, 70\}$ of ten elements. Let B be $\{60, 35, 73\}$ of three elements. Then, your answer should be the following: for 60, return 5 (because there 5 numbers in A smaller than 60); for 35, return 3; for 73, return 7.

   (a) Design an $O(n \log n)$ time algorithm for solving the problem.

   Answer: For problem (a), we first sort the array A in $O(n \log n)$ time and sort the array B in $O(m \log m)$ time. Because $m \leq n$, $m \log m = O(n \log n)$. Then, we scan the sorted array A and the sorted array B simultaneously, to compute the number of elements of A that is smaller than $B[i]$ for each i with $1 \leq i \leq m$. This can be done in $O(n + m)$ time by a scanning procedure similar to the procedure of merging two sorted lists. Note that $n + m = O(n)$. Hence, the total time of the algorithm is $O(n \log n)$, dominated by the sorting of A.

   (b) Design an $O(nm)$ time algorithm for the problem. Note that this is better than the $O(n \log n)$ time algorithm if $m < \log n$.

   Answer: For problem (b), we do not do any sorting this time. For each element $B[i]$, we simply do a linear scan on A to compute the number of elements of A smaller than $B[i]$, which takes $O(n)$ time. Hence, the total time is $O(nm)$.

   (c) Improve your algorithm to $O(n \log m)$ time by the divide and conquer technique. Because $m \leq n$, this is better than both the $O(n \log n)$ time and the $O(nm)$ time algorithms described above. Note that since $m \leq n$, you cannot sort the array A because that would take $O(n \log n)$ time, which is not $O(n \log m)$ as m may be much smaller than n.

   Answer: For problem (c), we only sort the array B, which takes $O(m \log m)$ time. Note that $m \log m = O(n \log m)$ as $m \leq n$. Then, we do a divide and conquer algorithm as follows. From now on, the array B is sorted. For each index $i \in [1, m]$, we will use $C[i]$ to store the number of elements of A smaller than $B[i]$. Therefore, our goal is to compute the array $C[1 \cdots m]$.

   We take the middle element $B[\frac{m}{2}]$ of B (which divides B into two subarrays $B[1 \cdots \frac{m}{2} - 1]$ and $B[\frac{m}{2} + 1 \cdots m]$ such that all elements of the first subarray are smaller than $B[\frac{m}{2}]$ and all elements of the second subarray are larger than $B[\frac{m}{2}]$).

   We process $B[\frac{m}{2}]$ as follows. By simply scanning the entire array A, we can compute the number of elements of A smaller than $B[\frac{m}{2}]$, and store that number in $C[\frac{m}{2}]$. In addition, we use $B[\frac{m}{2}]$ to partition A into two subarrays A_1 and A_2, such that A_1 contains all elements of A smaller than $B[\frac{m}{2}]$ and A_2 contains the rest.

   Next, we work on A_1 and $B[1 \cdots \frac{m}{2} - 1]$ recursively, and work on A_2 and $B[\frac{m}{2} + 1 \cdots m]$ recursively.

   Specifically, for the subproblem on A_1 and $B[1 \cdots \frac{m}{2} - 1]$, we take the middle element $B[\frac{m}{4}]$ of $B[1 \cdots \frac{m}{2} - 1]$ and process $B[\frac{m}{4}]$ as follows. By scanning A_1, we can compute the number of elements of A_1 smaller than $B[\frac{m}{4}]$, which is also the number of elements of A smaller than $B[\frac{m}{4}]$, and store that number in $C[\frac{m}{4}]$. Also, we use $B[\frac{m}{4}]$ to partition

A_1 into two subarrays with one containing all elements of A_1 smaller than $B[\frac{m}{4}]$ and the other containing the rest.

For the subproblem on A_2 and $B[\frac{m}{2}+1\cdots m]$, we take the middle element $B[\frac{3m}{4}]$ of $B[\frac{m}{2}+1\cdots m]$ and process $B[\frac{3m}{4}]$ as follows. By scanning A_2, we can compute the number of elements of A_2 smaller than $B[\frac{3m}{4}]$, and let k be that number. Observe that the number of elements of A smaller than $B[\frac{3m}{4}]$ is actually equal to k plus the number of elements in A_1 (let $|A_1|$ denote the size of $|A_1|$). Thus, we store the number $k+|A_1|$ in $C[\frac{m}{4}]$. This also means that when working on the subproblem of A_2 and $B[\frac{m}{2}+1\cdots m]$, we need to store the size of A_1. Also, we use $B[\frac{3m}{4}]$ to partition A_2 into two subarrays with one containing all elements of A_2 smaller than $B[\frac{3m}{4}]$ and the other containing the rest.

Note that the above processing $B[\frac{m}{4}]$ and $B[\frac{3m}{4}]$ takes $O(n)$ time in total. The above also obtained four subproblems on four subarrays of A and the four subarrays $B[1\cdots\frac{m}{4}-1]$, $B[\frac{m}{4}+1\cdots\frac{m}{2}-1]$, $B[\frac{m}{2}+1\cdots\frac{3m}{4}-1]$, $B[\frac{3m}{4}+1\cdots m]$ of B, respectively. We solve these problems recursively. The base case happens when a subarray of B contains only one element $B[i]$, in which case we only need to scan the corresponding subarray of A (in order to compute $C[i]$) without any further dividing.

The overall procedure of the algorithm can be considered as a tree structure partitioning the array B. The height of tree is $O(\log m)$ because B has m elements. For the running time, processing the elements of B at each level of the tree takes $O(n)$ time in total (because all scanning procedures at each level of the tree essentially scan the entire array A exactly once). Therefore, the total time of the algorithm is $O(n\log m)$.

Instead, we can also write the recurrence for the running time of the algorithm. Because we have two parameters n and m, we use $T(n,m)$ to denote the running time.

$$T(n,m)=\begin{cases}T(n_1,\frac{m}{2})+T(n-n_1,\frac{m}{2})+n & \text{if } m>1\\ n & \text{if } m=1\end{cases} \tag{1}$$

Here, n_1 is the number of numbers in the subarray A and $n-n_1$ is the number of numbers in the other subarray A_2. $\frac{m}{2}$ is the size of each of the two subarrays of B partitioned by $B[\frac{m}{2}]$. The processing of $B[\frac{m}{2}]$ takes $O(n)$ time (i.e., scanning the array A). If B has only one element (i.e., the case $m=1$), which is the base case, then after processing the only element, we do not need to further divide A. You may solve the recurrence by expanding, recursion tree, or guess-and-verification, to obtain $T(n,m)=O(n\log m)$.