CS5050 Advanced Algorithms

Spring Semester, 2018

Homework Solution 2

Haitao Wang

1. (a) The inversions are $(4,2),(4,1),(2,1),(9,1),(9,7)$.

   (b) The array with most inversions is $\langle n, n-1, n-2, \ldots, 2, 1\rangle$. The number of all inversions is $(n-1)+(n-2)+\cdots+2+1 = n(n-1)/2$.

   (c) The algorithm is similar to the merge-sort. The difference is that in the merge procedure (i.e., the combine step), in addition to merging two sorted lists, we also count the number of inversions. We have the following algorithm.

   If $n \leq 1$, we simply return $A[1]$ and we also return a value 0 that is the number of inversions in $A[1 \ldots n]$ (Since $n = 1$ and A has only one number, the number of inversions is zero). Otherwise, we perform the following steps.

   (1) Divide: Divide the array $A[1 \cdots n]$ into two sub-arrays $A[1 \cdots \frac{n}{2}]$ and $A[\frac{n}{2}+1 \cdots n]$. Let A_1 denote $A[1 \cdots \frac{n}{2}]$ and A_2 denote $A[\frac{n}{2}+1 \cdots n]$.

   (2) Conquer: Recursively sort the two sub-arrays A_1 and A_2, and compute the numbers of inversions in the two sub-arrays, respectively. Let I_1 be the number of inversions of the elements in A_1, and let I_2 be the number of inversions of the elements in A_2. Both I_1 and I_2 are computed in this step.

   (3) Combine: In Step (2), we have computed two sorted lists for A_1 and A_2, respectively; as in the merge-sort, we merge these two sorted lists into one sorted list for $A[1 \cdots n]$. In addition, we compute the number (denoted by I_3) of inversions $(A[i], A[j])$ such that $1 \leq i \leq \frac{n}{2}$ and $\frac{n}{2}+1 \leq j \leq n$; in other words, $A[i] > A[j]$ and $A[i]$ is in A_1 and $A[j]$ is in A_2. (The algorithm for computing I_3 will be given later.) Finally, return the sorted list of $A[1 \cdots n]$, and return the value $I_1 + I_2 + I_3$ as the number of inversions in $A[1 \cdots n]$.

   To see the correctness of the algorithm, we show below that the number of inversions in $A[1 \cdots n]$ is $I_1 + I_2 + I_3$. Consider any inversion $(A[i], A[j])$ in $A[1 \cdots n]$. Note that $i < j$. There are three cases for i and j: (1) $i < j \leq \frac{n}{2}$, which implies that both $A[i]$ and $A[j]$ are in A_1; (2) $\frac{n}{2}+1 \leq i < j$, which implies that both $A[i]$ and $A[j]$ are in A_2; (3) $i \leq \frac{n}{2} < j$, which implies that $A[i]$ is in A_1 and $A[j]$ is in A_2. Since I_1, I_2, and I_3 are the numbers of inversions in the above three cases, respectively, the number of inversions in $A[1 \cdots n]$ is $I_1 + I_2 + I_3$.

   It remains to design the combine step, in particular, to compute the value I_3. Consider any number $A[j]$ in A_2. For any number $A[i]$ in A_1, $(A[i], A[j])$ is an inversion if and only if $A[i] > A[j]$, and if $(A[i], A[j])$ is an inversion, we say that this version is *induced* by $A[j]$. Denote by x_j the number of inversions induced by $A[j]$. If we know x_j for each number $A[j]$ in A_2, then the sum of these x_j's is I_3. To compute the value x_j, it is easy

to see that x_j is equal to the number of elements in A_1 that are larger than $A[j]$. We can compute x_j by the following procedure.

If the number of elements in A_2 that are larger than $A[j]$ is y_j and the number of elements in $A[1 \dots n]$ that are larger than $A[j]$ is y'_j, then we have $x_j = y'_j - y_j$. Since we already have the sorted list for A_2 and the sorted list for $A[1 \dots n]$, the values y_j and y'_j for all elements $A[j]$ in A_2 can be easily computed in $O(n)$ time. Consequently, the values x_j for all elements $A[j]$ in A_2 can be computed in $O(n)$ time. The value I_3 can then be computed by summing up all x_j's.

In summary, the combine step can be implemented in $O(n)$ time, which is the same as the merge-sort. Therefore, the running time of the entire algorithm is $O(n \log n)$, the same as the merge sort.

2. The three algorithms are given below.

Question (a). 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.

Question (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 $O(nm)$.

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

We *process* $B[m/2]$ as follows. By simply scanning the entire array A, we can compute the number of elements of A smaller than $B[m/2]$, and store that number in $C[m/2]$. In addition, we use $B[m/2]$ to partition A into two subarrays A_1 and A_2, such that A_1 contains all elements of A smaller than $B[m/2]$ 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 $B[m/4]$ of $B[1 \cdots \frac{m}{2} - 1]$ and process $B[m/4]$ as follows. By scanning A_1, we can compute the number of elements of A_1 smaller than $B[m/4]$, which is also the number of elements of A smaller than $B[m/4]$, and store that number in $C[m/4]$. Also, we use $B[m/4]$ to partition A_1 into

two subarrays with one containing all elements of A_1 smaller than $B[m/4]$ 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 $B[3m/4]$ of $B[\frac{m}{2}+1\cdots m]$ and process $B[3m/4]$ as follows. By scanning A_2, we can compute the number of elements of A_2 smaller than $B[3m/4]$, and let k be that number. Observe that the number of elements of A smaller than $B[3m/4]$ 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 $C[m/4]$. 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 $B[3m/4]$ to partition A_2 into two subarrays with one containing all elements of A_2 smaller than $B[3m/4]$ and the other containing the rest.

Note that the above processing $B[m/4]$ and $B[3m/4]$ takes $O(n)$ time in total. The above also obtained four subproblems on four subarrays of A and the four subarrays $B[1\cdots m/4-1]$, $B[m/4+1\cdots m/2-1]$, $B[m/2+1\cdots 3m/4-1]$, $B[3m/4+1\cdots m]$ 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.

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

Here, n_1 is the number of numbers in the subarray A_1 and $n-n_1$ is the number of numbers in the other subarray A_2. $m/2$ is the size of each of the two subarrays of B partitioned by $B[m/2]$. The processing of $B[m/2]$ 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 $T(n,m)=O(n\log m)$.

3. (a) $T(n)=2\cdot T(\frac{n}{2})+n^3$.

If we keep expanding the recurrence, we have

$$\begin{aligned}T(n)&=2\cdot T(\frac{n}{2})+n^3\\&=4\cdot T(\frac{n}{4})+\frac{n^3}{2^3}+n^3\\&=8\cdot T(\frac{n}{8})+\frac{n^3}{4^3}+\frac{n^3}{2^3}+n^3\\&=\cdots\\&=2^i\cdot T(\frac{n}{2^i})+(\frac{1}{2^3})^{i-1}\cdot n^3+\cdots+(\frac{1}{2^3})^2\cdot n^3+\frac{1}{2^3}\cdot n^3+n^3\end{aligned}$$

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

$$\begin{aligned} T(n) &= 2^k \cdot T(\frac{n}{2^k}) + (\frac{1}{2^3})^{k-1} \cdot n^3 + \cdots + (\frac{1}{2^3})^2 \cdot n^3 + \frac{1}{2^3} \cdot n^3 + n^3 \\ &= 2^k \cdot T(1) + n^3 \cdot [(\frac{1}{2^3})^{k-1} + \cdots + (\frac{1}{2^3})^2 + \frac{1}{2^3} + 1] \end{aligned}$$

Note that $2^k = n$ and $T(1) = O(1)$. Recall that $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ holds for $|x| < 1$. We have $(\frac{1}{2^3})^{k-1} + \cdots + (\frac{1}{2^3})^2 + \frac{1}{2^3} + 1 \leq \frac{1}{1-\frac{1}{2^3}} = \frac{8}{7}$.

Therefore, we obtain $T(n) \leq n \cdot O(1) + n^3 \cdot \frac{8}{7} = O(n^3)$.

Alternatively, we can use Master theorem with $a = b = 2$ and $f(n) = n^3$ (recall the form $T(n) = a \cdot T(n/b) + f(n)$). Hence, $\log_b a = 1$ and $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon = 1$. To see whether Case 3 of Master theorem can be applied, we also need to check whether there exist a constant $c < 1$ and $n_0 \geq 1$, such that $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ for all $n \geq n_0$.

Suppose we let $c = 0.5$ and $n_0 = 1$. Then, $af(n/b) = 2 \cdot (\frac{n}{2})^3 = \frac{n^3}{4} \leq c \cdot f(n) = 0.5 \cdot n^3$ for any $n \geq n_0$. Therefore, Case 3 of Master theorem can be applied and we obtain $T(n) = \Theta(f(n))$, which is $\Theta(n^3)$.

(b) $T(n) = 4 \cdot T(\frac{n}{2}) + n\sqrt{n}$.

The easiest way is to use Master theorem, with $a = 4$, $b = 2$, and $f(n) = n^{1.5}$. Hence, $\log_b a = 2$ and we have $f(n) = n = O(n^{\log_b a - \epsilon})$ with $\epsilon = 0.25$. Thus, we can apply Case 1 of Master theorem to obtain $T(n) = O(n^{\log_b a})$, which is $O(n^2)$.

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

For this problem, none of the cases of Master theorem can apply. I will give two approaches for this problem: expanding the recurrence and guess-and-verification.

By expanding the recurrence, we have

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

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

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

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 $T(n) = n\cdot O(1) + n\log n\cdot\frac{\log n+1}{2} = O(n\log^2 n)$.
Next, we use the guess-and-verification approach.
We guess $T(n) = O(n\log^2 n)$. In other words, we want to prove that there exit a constant c and $n_0 \geq 1$, such that $T(n) \leq cn\log^2 n$ holds for all $n \geq n_0$.
We assume the above inequality holds for $n/2$, i.e., $T(n/2) \leq c(n/2)\log^2(n/2)$. Then, we deduce the following

$$\begin{aligned}
T(n) &= 2\cdot T(\frac{n}{2}) + n\log n \\
&\leq 2\cdot c\cdot\frac{n}{2}\cdot\log^2\frac{n}{2} + n\log n
\end{aligned}$$

Note that $\log\frac{n}{2} = \log n - 1$, and thus, $\log^2\frac{n}{2} = (\log n - 1)^2 = \log^2 n - 2\log n + 1$. Therefore, we have

$$\begin{aligned}
T(n) &\leq cn\cdot(\log^2 n - 2\log n + 1) + n\log n \\
&= cn\log^2 n - 2cn\log n + cn + n\log n
\end{aligned}$$

Our goal is to find c and $n_0 \geq 1$ such that $T(n) \leq cn\log^2 n$ holds for all $n \geq n_0$. Now that $T(n) \leq cn\log^2 n - 2cn\log n + cn + n\log n$, to prove $T(n) \leq cn\log^2 n$, it is sufficient to prove $-2cn\log n + cn + n\log n \leq 0$, or equivalently to prove $cn + n\log n \leq 2cn\log n$.
If we let $c = 1$ and $n_0 = 2$, then clearly $cn + n\log n \leq 2cn\log n$ holds for all $n \geq n_0$.
We conclude that our guess that $T(n) = O(n\log^2 n)$ is correct.

(d) $T(n) = T(\frac{3}{4}\cdot n) + n$.
We can prove $T(n) = O(n)$ by using Case 3 of Master theorem, with $a = 1$, $b = \frac{4}{3}$, and $f(n) = n$. It is also easy to use the expanding-recurrence approach since $a = 1$, as follows.
By expanding the recurrence, we can obtain $T(n) = n + \frac{3}{4}n + (\frac{3}{4})^2 n + (\frac{3}{4})^3 n + \cdots$. Recall that $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ holds for $|x| < 1$. We have $1 + \frac{3}{4} + (\frac{3}{4})^2 + (\frac{3}{4})^3 + \cdots = \frac{1}{1-\frac{3}{4}} = 4$.
Hence, $T(n) = O(n)$.

4. Essentially we can mode the problem as following: Given an array $A[1,\ldots,n]$ of n elements (i.e., $A[i] = p(i)$ for each i), find a pair of indices (i,j) with $i < j$ such that $A[j] - A[i]$ is maximized among all such pairs (i,j).

   Suppose (i^*, j^*) is the optimal solution pair. If we divide the array A into two subarrays $A_1 = A[1,\ldots,\frac{n}{2}]$ and $A_2 = A[\frac{n}{2}+1, n]$, then an *observation* is that one of the three cases

must happen: (1) both i^* and j^* are in A_1; (2) both i^* and j^* are in A_2; (3) i^* is in A_1 and j^* is in A_2. The solutions in the first two cases can be found by recursively solving the sub-problems on A_1 and A_2. To find the solution for the third case, essentially we want to find an index i with $1 \leq i \leq \frac{n}{2}$ and an index j with $\frac{n}{2} + 1 \leq j \leq n$ such that $A[j] - A[i]$ is maximized, and such a pair (i, j) can be obtained by finding the smallest element $A[i]$ in A_1 and the largest element $A[j]$ in A_2.

Thus, using the divide-and-conquer approach, our algorithm is to take the best of the following three possible solutions:

- The optimal solution on A_1.
- The optimal solution on A_2.
- The maximum of $A[j] - A[i]$, over $A[i] \in A_1$ and $A[j] \in A_2$.

If $T(n)$ is the overall running time of the algorithm, the first two solutions are computed in time $T(n/2)$, each by recursion, and the third solution is computed by finding the smallest element in A_1 and the largest element in A_2, which takes time $O(n)$. Thus, we have the following recurrence for the running time: $T(n) = 2 \cdot T(n/2) + O(n)$. Solving the recurrence, we can obtain $T(n) = O(n \log n)$.

Remark 1. Suppose (i, j) is the pair returned by the algorithm; if it turns out that $A[i] > A[j]$, then this means that there is no way to make money during the n days.

Remark 2. If we maintain the smallest and the largest elements in the subarrays A_1 and A_2 (in the similar way as in the problem of finding the largest number in an array discussed in class), then finding the above third solution only takes $O(1)$ time. In this way, the combine step of the algorithm runs in $O(1)$ time, and the recurrence becomes $T(n) = 2 \cdot T(n/2) + O(1)$. Solving the recurrent gives us $T(n) = O(n)$.

The second approach. Here is another approach to solve the problem by reducing it to the maximum sub-array problem we studied in class. We first create an array $B[1, \ldots, n-1]$ of $n-1$ elements as follow: for each $1 \leq i \leq n-1$, set $B[i] = A[i+1] - A[i]$. The *key observation* is: If $B[i \cdots j]$ is a maximum sub-array of B, then $(i, j+1)$ is an optimal pair for our original problem on the array A. Then, we can apply the algorithm for computing the maximum sub-array to solve the problem.