Analysis of Algorithms -Final (Solutions)

K. Subramani
LCSEE,
West Virginia University,
Morgantown, WV
{ksmani@csee.wvu.edu}

1 Problems

1. Characterize the following recurrence: (5 points)

$$\begin{aligned} T(n) &= 1, \ if\ n = 1, \\ &= 9 \cdot T(\frac{n}{3}) + n^3 \cdot \log n, \ \ if\ n \geq 2 \end{aligned}$$

Solution: Using the notation of the Master Theorem, we have, $a = 9$, $b = 3$, $c = 1$, $d = 2$ and $f(n) = n^3 \cdot \log n$. Hence, $n^{\log_b a} = n^{\log_3 9} = n^2$. Note that $n^3 \cdot \log n$ is $\Omega(n^{2+\epsilon})$, for $\epsilon = 0.5$ (say) and

$$\begin{aligned} 9 \cdot (\frac{n}{3})^3 \cdot \log \frac{n}{3} &= 9 \cdot \frac{n^3}{27} \cdot (\log n - \log 3) \\ &= \frac{1}{3} \cdot n^3 \log n - \frac{1}{3} \cdot \log 3 \cdot n^3 \\ &\leq \frac{1}{3} \cdot n^3 \log n, \ n \geq 2 \end{aligned}$$

Thus, we can choose $\delta = \frac{1}{3}$ and apply case (3) of the Master Theorem, to conclude that $T(n) = \Theta(n^3 \log n)$. □

2. In the deterministic implementation of the Quicksort algorithm, we choose the last item as the pivot, at each level of the recursion. Assume that there exists an oracle, which when given an array of n elements, returns the middle element, i.e., the element of rank $\frac{n}{2}$, in $O(n)$ time. Now, instead of choosing the last element of the array as the pivot, suppose that we use the element of rank $\frac{n}{2}$ as the pivot in each level of the recursion. (You may assume that at each level, the number of elements is divisible by 2.) What is the running time of this modified version of Quicksort, on an array that is already sorted? (5 points)

Solution: The key observation is that in $O(n)$ time, the array is broken into 2 parts of equal size. Thus, if $T(n)$ describes the running time of the algorithm on an input of n elements, then

$$\begin{aligned} T(n) &= 1, \ if\ n = 2 \\ &= 2 \cdot T(\frac{n}{2}) + O(n), \ otherwise \end{aligned}$$

Applying the Master Theorem, we conclude that $T(n) = \Theta(n \log n)$. It is important to note that giving a pre-sorted array does not affect the running time of the modified version of the Quicksort algorithm. If, on the other hand, we choose the last element of the array as the pivot, then the pre-sorted array would cause the algorithm to run in time $\Theta(n^2)$. □

3. Let $\mathbf{A}$ be an array of n *distinct* elements. If $i < j$ and $A[i] > A[j]$, for some indices i and j, then (i, j) is said to be an *inversion pair*.

   (a) List the inversion pairs of the array $\mathbf{A} = [2, 3, 8, 6, 1]$. (2 points)

   (b) Design a divide-and-conquer algorithm that outputs the *number of inversion pairs*, given an array having n elements. Analyze the time and space requirements of your algorithm. (8 points)
   Hint: Modify Merge-Sort.

Solution:

(a) The inversion pairs of $\mathbf{A}$ are: $(1, 5)$, $(2, 5)$, $(3, 4)$, $(3, 5)$ and $(4, 5)$. Note that the inversion pairs are $2-$tuples of array indices and not $2-$tuples of array elements!

(b) Algorithm (1.1) represents a divide-and-conquer algorithm for the inversion-pair problem; note that the algorithm also sorts the input array $\mathbf{A}$.

Function NUMBER-INVERSION-PAIRS($\mathbf{A}$, low, $high$)
1: **if** $(low < high)$ **then**
2: $mid = \frac{low+high}{2}$
3: c_l = NUMBER-INVERSION-PAIRS($\mathbf{A}$, low, mid)
4: c_u = NUMBER-INVERSION-PAIRS($\mathbf{A}$, $mid + 1$, $high$)
5: c_m = MERGE-INVERSION-PAIRS($\mathbf{A}$, low, mid, $high$)
6: **return**$(c_l + c_u + c_m)$
7: **else**
8: **return**(0)
9: **end if**

Algorithm 1.1: Algorithm for determining the number of inversion pairs in the sub-array $\{A[low], A[low + 1], \ldots, A[high]\}$

1.1 Correctness

We prove the correctness of Algorithm (1.1), by using induction on the number of elements in the array $\mathbf{A}$. Note that the number of elements in $\{A[low], A[low + 1], \ldots, A[high]\}$ is $(high - low + 1)$ and is denoted by $|\mathbf{A}|$.

i. Observe that the number of inversion pairs in $\mathbf{A}$ is 0, when $|\mathbf{A}| \leq 1$. When $\mathbf{A}$ has at most one element, $(high \leq low)$ and Step (8) of Algorithm (1.1) is executed. It follows that Algorithm (1.1) works correctly when $|\mathbf{A}| \leq 1$.

ii. Assume that Algorithm (1.1) returns the correct number of inversion pairs when $|\mathbf{A}| \leq k$, for all $k = 2, 3, \ldots n - 1$ and also sorts $\mathbf{A}$. Now consider the case, when $|\mathbf{A}| = n$. Step (2) of Algorithm (1.1) breaks up the array into 2 sub-arrays. Let $\mathbf{A_l}$ denote the sub-array $\{A[low], A[low+1], \ldots, A[mid]\}$ and $\mathbf{A_h}$ denote the sub-array $\{A[mid+1], A[mid+2], \ldots, A[high]\}$. Clearly $|\mathbf{A_l}| < n$ and $|\mathbf{A_h}| < n$. Hence, we can apply the inductive hypothesis to conclude that the number of inversion pairs computed in Steps (3) and (4) are correct, i.e., c_l stores the correct number of inversion pairs in $\mathbf{A_l}$ and c_h stores the correct number of inversion pairs in $\mathbf{A_h}$. Further $\mathbf{A_l}$ and $\mathbf{A_h}$ are sorted.
We now analyze the MERGE-INVERSION-PAIRS() procedure. An element of $\mathbf{A_l}$ that is moved into $\mathbf{C}$, does not form inversion pairs with any of the elements in $\{A_h[q], A_h[q + 1], \ldots, A_h[high]\}$. However, an element of $\mathbf{A_h}$ that is moved into $\mathbf{C}$ forms inversion pairs with all the elements in $\mathbf{A_l}$ that have not been processed, i.e., $\{A_l[p], A_l[p+1], \ldots, A_l[mid]\}$. Since we are only concerned with the number of inversion pairs, we update the inversion pair count by $(mid - p + 1)$. An inversion pair (i, j) in $\{A[low], A[low + 1], \ldots, A[high]\}$ must have one of the following forms:

A. $i < j$, $A[i] > A[j]$ and $i, j \in \{low, low + 1, \ldots, mid\}$ - This case is recursively handled (Step (3) of Algorithm (1.1));

```
Function MERGE-INVERSION-PAIRS(A, low, mid, high)
 1: {We assume that the sub-arrays {A[low], A[low + 1], ..., A[mid]} and {A[mid + 1], A[mid + 2], ..., A[high]}
    have been sorted.}
 2: p = low; q = mid + 1; index = low; c_m = 0.
 3: while (p ≤ mid) and (q ≤ high) do
 4:    if (A[p] ≤ A[q]) then
 5:       C[index] = A[p]
 6:       {A[p] does not form an inversion pair with an element of A_h.}
 7:       p++
 8:    else
 9:       C[index] = A[q]
10:       {A[q] forms an inversion pair with each of the elements {A[p], A[p + 1], ..., A[mid]} }
11:       q++
12:       c_m += (mid − p + 1)
13:    end if
14:    index++
15: end while
16: if (p > mid) then
17:    {All the elemtents in A_l have been processed; copy the elements in A_h into C.}
18:    for (i = q to high) do
19:       C[index] = A[i]
20:       index++
21:    end for
22: else
23:    {All the elements in A_h have been processed; copy the elements in A_l into C.}
24:    for (i = p to mid) do
25:       C[index] = A[i]
26:       index++
27:    end for
28: end if
29: {Now copy the elements in C back into A}
30: for (i = low to high) do
31:    A[i] = C[i]
32: end for{{A[low], A[low + 1], ..., A[high]} is now in sorted order}
33: return(c_m)
```

Algorithm 1.2: The Merge Procedure

B. $i < j,\ A[i] > A[j]$ and $i, j \in \{mid+1, mid+2, \ldots, high\}$ - This case is also recursively handled (Step (4) of Algorithm (1.1));

C. $i < j,\ A[i] > A[j],\ i \in \{low, low+1, \ldots, mid\}$ - and $j \in \{mid+1, mid+2, \ldots, high\}$ - This case is handled by the MERGE-INVERSION-PAIRS() procedure.

Thus, we have accounted for all the inversion pairs in $\{A[low], A[low+1], \ldots, A[high]\}$, thereby proving that if Algorithm (1.1) is correct when $|\mathbf{A}| = k,\ k = 2, 3, \ldots, n-1$, then it must be correct for $|\mathbf{A}| = n$. Applying the principle of Mathematical induction, we conclude that Algorithm (1.1) is correct.

1.2 Analysis

Observe that all that we need in terms of extra space is the merge array $\mathbf{C}$. Thus Algorithm (1.1) requires $\Theta(n)$ space.

Let $T(n)$ denote the running time of Algorithm (1.1) on an array of n elements. It is not hard to see that (exactly as in MERGE-SORT())

$$\begin{aligned} T(n) &= 1,\ if\ n = 1 \\ &= 2 \cdot T(\frac{n}{2}) + \Theta(n),\ otherwise \end{aligned}$$

It follows that $T(n) = \Theta(n \log n)$, as per the Master Theorem.

□

4. Let $\mathbf{A} =< a_1, a_2, \ldots, a_n >$ be a sequence of integers, where each a_i can be positive, negative or zero. We define a con-sequence of $\mathbf{A}$ as a subsequence of $\mathbf{A}$, with all the entries appearing consecutively in $\mathbf{A}$. For instance, if $\mathbf{A} =< 7, -4, 3, 5, -2 >$, $< -4, 3, 5 >$ is a con-sequence. Note that $< -4, 5, -2 >$ is *not* a con-sequence, although it is a subsequence of $\mathbf{A}$. The *worth* of a con-sequence is defined as the sum of its elements, e.g., the worth of $< -4, 3, 5 >$ is 4. Devise an algorithm that, given a sequence $\mathbf{A}$ of n elements, outputs the con-sequence of maximum worth. For instance, the con-sequence of $\mathbf{A}$ having maximum worth is $< 7, -4, 3, 5 >$ and it has worth 11. Analyze the time and space requirements of your algorithm. (10 points)
Hint: Use Dynamic Programming.

Solution: Let $\mathcal{A} =< a_i, a_{i+1}, \ldots, a_j >,\ 1 \leq i \leq j \leq n$ denote the optimal con-sequence in $\mathbf{A} =< a_1, a_2, \ldots, a_n >$. Observe that $\mathcal{A}' =< a_{i+1}, a_{i+2}, \ldots, a_j >$ *must* be the optimal con-sequence in $\mathbf{A}$ over all con-sequences starting at a_{i+1}; if this were not the case and there existed a con-sequence $\mathcal{A}'' =< a_{i+1}, a_{i+2}, \ldots, a_k >$ of greater worth than $\mathcal{A}'$, then this con-sequence could be combined with a_i to get a con-sequence of worth exceeding the worth of $\mathcal{A}$, thereby contradicting the optimality of $\mathcal{A}$. Thus, the principle of optimality applies and we can use Dynamic Programming.

Let $w[i]$ denote the worth of the maximum con-sequence $\mathbf{A}$ which begins at a_i. The crucial observation is that if $w[i+1] < 0$, then $w[i] = a_i$. Note that any con-sequence beginning at a_{i+1} has negative worth, since $w[i+1]$ represents the con-sequence of maximum worth. Thus, including a_{i+1} in any con-sequence starting at a_i, will only decrease the worth of the sequence $< a_i >$.

Accordingly, we have,

$$\begin{aligned} w[i] &= a_i + w[i+1],\ 1 \leq i < n\ and\ w[i+1] \geq 0 \\ &= a_i,\ otherwise \end{aligned}$$

□

2 Analysis

It is not hard to see that the algorithm runs in $\Theta(n)$ space and takes $\Theta(n)$ time.

```
Function MaxWorth Con-sequence(A, n)
 1: for (i = 1 to n) do
 2:    w[i] = a_i
 3: end for
 4: for (i = (n − 1) downto 1) do
 5:    if (w[i + 1] ≥ 0) then
 6:       w[i] = a_i + w[i + 1]
 7:    end if
 8: end for
 9: {Now find the index at which the maximum worth con-sequence starts}
10: max = 1
11: for (i = 2 to n) do
12:    if (w[max] < w[i]) then
13:       max = i
14:    end if
15: end for
16: {Print out the con-sequence of maximum worth!}
17: t = w[max]
18: while (t ≠ 0) do
19:    print a_max
20:    max + +
21:    t = t − a_max
22: end while
```

Algorithm 1.3: Algorithm for determining the maximum worth of a con-sequence