Advanced Analysis of Algorithms - Midterm (Solutions)

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

1 Problems

1. Solve the following recurrence using substitution:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2 \cdot T(\frac{n}{2}) + n \cdot \log n, \; n \geq 2 \end{aligned}$$

   Solution: Putting $n = 2^k$, the recurrence relation can be written as:

$$\begin{aligned} T(2^0) &= 0 \\ T(2^k) &= 2 \cdot T(2^{k-1}) + k \cdot 2^k, \; k \geq 1 \end{aligned}$$

   Now note that

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

   From this it follows, that

$$T(n) = n \cdot \frac{(\log n) \cdot (\log n + 1)}{2}$$

   Now try solving the problem, using the Master Theorem! □

2. Given an array $\mathbf{A}$ of n integer elements, design an algorithm that computes the number of inversion pairs and runs in $O(n \cdot \log n)$ time. Note that an inversion pair is a pair of indices (i, j), such that $i < j$ and $\mathbf{A}[i] > \mathbf{A}[j]$.
   Hint: Use Divide-and-Conquer.

   Solution: Algorithm (1.1) represents a divide-and-conquer algorithm for the inversion-pair problem; note that the algorithm also sorts the input array $\mathbf{A}$. Although you were not required to provide a correctness proof, I have provided one for your benefit.

```
Function NUMBER-INVERSION-PAIRS($\mathbf{A}, low, high$)
if ($low < high$) then
    $mid = \frac{low+high}{2}$
    $c_l$ = NUMBER-INVERSION-PAIRS($\mathbf{A}, low, mid$)
    $c_u$ =NUMBER-INVERSION-PAIRS($\mathbf{A}, mid+1, high$)
    $c_m$ =MERGE-INVERSION-PAIRS($\mathbf{A}, low, mid, high$)
    return($c_l + c_u + c_m$)
else
    return(0)
end if
```

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

```
Function MERGE-INVERSION-PAIRS($\mathbf{A}, low, mid, high$)
{We assume that the sub-arrays $\{A[low], A[low+1], \ldots, A[mid]\}$ and $\{A[mid+1], A[mid+2], \ldots, A[high]\}$ have been sorted.}
$p = low$; $q = (mid+1)$; $index = low$; $c_m = 0$.
while ($p \le mid$) and ($q \le high$) do
    if ($A[p] \le A[q]$) then
        $C[index] = A[p]$
        {$A[p]$ does not form an inversion pair with an element of $\mathbf{A_h}$.}
        $p++$
    else
        $C[index] = A[q]$
        {$A[q]$ forms an inversion pair with each of the elements $\{A[p], A[p+1], \ldots, A[mid]\}$}
        $q++$
        $c_m += (mid - p + 1)$
    end if
    $index++$
end while
if ($p > mid$) then
    {All the elements in $\mathbf{A_l}$ have been processed; copy the elements in $\mathbf{A_h}$ into $\mathbf{C}$.}
    for ($i = q$ to $high$) do
        $C[index] = A[i]$
        $index++$
    end for
else
    {All the elements in $\mathbf{A_h}$ have been processed; copy the elements in $\mathbf{A_l}$ into $\mathbf{C}$.}
    for ($i = p$ to $mid$) do
        $C[index] = A[i]$
        $index++$
    end for
end if
{Now copy the elements in $\mathbf{C}$ back into $\mathbf{A}$.}
for ($i = low$ to $high$) do
    $A[i] = C[i]$
end for
{$\{A[low], A[low+1], \ldots, A[high]\}$ is now in sorted order}
return($c_m$)
```

Algorithm 1.2: The Merge Procedure.

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 $|\mathbf{A}|$.

(a) 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 $|\mathbf{A}| \leq 1$.

(b) Assume that Algorithm (1.1) returns the correct number of inversion pairs when $|\mathbf{A}| \leq k$, $\forall k, k = 2, 3, \ldots, (n-1)$ 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 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:

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

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

   iii. $i < j$, $A[i] > A[j]$, $i \in \{low, (low+1), \ldots, mid\}$ - 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

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())

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

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

3. Construct the optimal binary search tree on the following four ordered keys, $key_1 \leq key_2 \leq key_3 \leq key_4$, with probability distribution $p_1 = \frac{1}{2}$, $p_2 = \frac{1}{8}$, $p_3 = \frac{1}{8}$ and $p_4 = \frac{1}{4}$.

Solution: From the algorithm given in [NN04], it is not hard to see that the following tree is optimal.

(i) key_1 at the root,

(ii) key_4 to the right of key_1 on level 1,

(iii) key_3 to the left of key_4 on level 2, and

(iv) key_2 to the left of key_3 on level 3.

□

4. In the Fractional Knapsack problem, you are given n objects $O = \{o_1, o_2, \ldots, o_n\}$ with respective weights $W = \{w_1, w_2, \ldots, w_n\}$ and respective profits $P = \{p_1, p_2, \ldots, p_n\}$. The goal is to pack these objects into a knapsack of capacity M, such that the profit of the objects in the knapsack is maximized, while the weight constraint is not violated. You may choose a fraction of an object, if you so decide; if $\alpha_i,\ 0 \le \alpha_i \le 1$ of object o_i is chosen, then the profit contribution of this object is $\alpha_i \cdot o_i$ and its weight contribution is $\alpha_i \cdot w_i$. Design a greedy algorithm for this problem and argue its correctness.

   Solution: The solution technique consists of the following steps:

   (i) Order the objects by profit per unit weight, so that $\frac{p_1}{w_1} \ge \frac{p_2}{w_2} \ge \ldots \frac{p_n}{w_n}$.

   (ii) Process the objects from o_1 to o_n. Pack as much as possible of o_1 in the knapsack. If the knapsack is full stop; otherwise, o_1 is included as a whole and there is weight capacity left over. Then pack as much as possible of o_2 in the knapsack and so on.

   Let $X =< x_1, x_2, \ldots, x_n >$ denote the greedy solution vector, where $x_i,\ 0 \le x_i \le 1$ is the fraction of o_i that is included in the knapsack. As per the description of the greedy algorithm, 0 or more of the $x_i s$ will be 1, followed by a fractional quantity, followed by $0s$. Let j be the first index such that $x_j \ne 1$. Then $x_i = 1,\ i = 1, 2, \ldots, j-1$ and $x_i = 0,\ i = j+1, j+2, \ldots, n$. Let $Y =< y_1, y_2, \ldots, y_n >$ denote an arbitrary optimal solution vector. We will show that Y can be gradually transformed into X, without decreasing profitability, while maintaining feasibility.

   We assume that $\sum_{i=1}^{n} w_i \cdot y_i = M$, since otherwise, we could pack more (of) objects into the knapsack, thereby proving that Y is sub-optimal. From the mechanics of the greedy algorithm, either $\sum_{i=1}^{n} w_i \cdot x_i = M$ or $X =< 1, 1, \ldots, 1 >$. In the latter case, X must be optimal, so there is nothing to be proved.

   Let k be the first index, where $x_k \ne y_k$. It must be the case that $x_k > y_k$. If $k < j$, then $x_k = 1$ and $x_k \ne y_k$ implies that $y_k < x_k$. If $k \ge j$ and $y_k > x_k$, then $\sum_{i=1}^{n} w_i \cdot y_i > M$, and knapsack feasibility is violated.

   Now increase y_k till it becomes x_k, while decreasing some or all of the $y_i s, i = k+1, \ldots, n$, so that the total weight in the knapsack stays the same. Let $Z =< z_1, z_2, \ldots, z_n >$ denote the new solution. Observe that $w_k \cdot (z_k - y_k) = \sum_{i=k+1}^{n} w_i \cdot (y_i - z_i)$, in order to maintain feasibility.

   Now,

$$
\begin{aligned}
\sum_{i=1}^{n} p_i \cdot z_i &= \sum_{i=1}^{n} p_i \cdot y_i + p_k \cdot (z_k - y_k) - \sum_{i=k+1}^{n} p_i \cdot (y_i - z_i) \\
&= \sum_{i=1}^{n} p_i \cdot y_i + p_k \cdot (z_k - y_k) \cdot \frac{w_k}{w_k} - \sum_{i=k+1}^{n} p_i \cdot (y_i - z_i) \cdot \frac{w_i}{w_i} \\
&\ge \sum_{i=1}^{n} p_i \cdot y_i + \frac{p_k}{w_k} \cdot (z_k - y_k) \cdot w_k - \sum_{i=k+1}^{n} \frac{p_k}{w_k} \cdot (y_i - z_i) \cdot w_i \\
&= \sum_{i=1}^{n} p_i \cdot y_i + \frac{p_k}{w_k} \cdot [(z_k - y_k) \cdot w_k - \sum_{i=k+1}^{n} w_i \cdot (z_i - y_i)] \\
&= \sum_{i=1}^{n} p_i \cdot y_i
\end{aligned}
$$

   Thus, Z is one step closer to X than Y is; arguing in this fashion, we can gradually transform Y into X, while maintaining feasibility and not decreasing profitability. This proves that the greedy solution is optimal. □

5. Argue that Randomized Quicksort takes $O(n \cdot \log n)$ comparisons, in the expected case, to sort an array of n elements.

   Solution: As per the algorithm in class, the pivot is chosen uniformly and at random from the array. The elements of the array are then partitioned about this pivot element and the algorithm either terminates on a partition (if there is at most one element in it) or recurses on it (if there are at least two elements in it).

Consider the computation tree created by a run of Randomized Quicksort on an array of n elements. The root of the tree has n elements and the leaves have exactly one element each. A node in this tree is called "good", if *both* its children have size at most $\frac{3}{4}^{th}$ of its size.

Claim 1.1 *A root to leaf path cannot have more than* $\log_{\frac{4}{3}} n$ *good nodes.*

Proof: Let L be a good node and let L' denote one of its two children. We must have $|L'| \leq \frac{3}{4}|L|$. Accordingly, if there are k good nodes on a root to leaf path, we have $(\frac{3}{4})^k \cdot n \leq 1$, which implies that $k \leq \log_{\frac{4}{3}} n$. $\Box$

Claim 1.2 *The expected height of the computation tree is* $O(\log n)$.

Proof: The expected height of the computation tree is the expected length of a randomly chosen root to leaf path. Focus on a specific root to leaf path of the computation tree, say p. At any node on p, Randomized Quicksort picks a pivot, uniformly and at random over all the elements in that node. Therefore, probability that a node is good is at least $\frac{1}{2}$. Now the expected number of nodes on p (which is the expected length of p)is equal to the number of recursive invocations made, before $\log_{\frac{4}{3}} n$ good nodes are created. However, for $\log_{\frac{4}{3}} n$ good nodes to be created, at most $2 \cdot \log_{\frac{4}{3}} n$ recursive invocations need to be made, as per the theorem given in class. Hence, the expected height of the computation tree is $O(\log n)$. $\Box$

Observe that the total number of comparisons in each level of the computation tree is $O(n)$ and hence the total number of comparisions performed by Randomized Quicksort is $O(n \cdot \log n)$, in the expected case. $\Box$

References

[NN04] Richard Neapolitan and Kumarss Naimipour. *Foundations of Algorithms Using C++ Pseudocode*. Jones and Bartlett, 2004.