Analysis of Algorithms - Midterm (Solutions)

L. Kovalchick
LCSEE,
West Virginia University,
Morgantown, WV
{lynn@csee.wvu.edu}

1. Consider the recurrence relation (6 points):

\[
T(1) = 1 \\
T(n) = 2 \cdot T(n-1) + 1, \ n > 1
\]

Show that \( T(n) = 2^n - 1 \)

**Proof:** Using induction:

**Base case \( T(1) \):**

\[
T(1) = 1 \\
T(1) = 2^1 - 1 \\
= 2 - 1 \\
= 1
\]

Thus, the base case is true.

Let us assume that \( T(k) \) is true, i.e.,

\[
T(k) = 2^k - 1
\]

We need to show that \( T(k+1) \) is true.

\[
T(k + 1) = 2 \cdot T(k + 1 - 1) + 1 \\
= 2 \cdot T(k) + 1 \\
= 2 \cdot (2^k - 1) + 1 \ (using \ the \ inductive \ hypothesis) \\
= 2^{k+1} - 2 + 1 \\
= 2^{k+1} - 1 \\
T(k + 1) = 2^{k+1} - 1
\]

Thus, \( P(k+1) \) is true and we have shown that \( P(k) \rightarrow P(k + 1) \); applying the principle of mathematical induction, we conclude that the conjecture is true. \( \Box \)
2. Show that if \( f(n) = O(g(n)) \) and \( e(n) = O(h(n)) \), then \( f(n) \cdot e(n) = O(g(n) \cdot h(n)) \). (4 points)

**Proof:** By definition of \('O', f(n) = O(g(n)) \) implies that:
\[
f(n) \leq c \cdot g(n)
\]

Also, by definition of \('O', e(n) = O(h(n)) \) implies that:
\[
e(n) \leq c' \cdot h(n)
\]

Observe that:
\[
f(n) \cdot e(n) \leq c \cdot g(n) \cdot c' \cdot h(n)
\]
\[
\leq c'' \cdot g(n) \cdot h(n)
\]

Then, by definition of \('O', f(n) \cdot e(n) = O(g(n) \cdot h(n)). \ □

3. Let \( T \) be a proper binary tree of height \( h \), having \( n \) nodes. Show that \( h \geq \log_2(n+1) - 1 \). (6 points)

**Proof:** Note that we want to find a lower bound on the height \( h \) of a proper binary tree containing \( n \) nodes. The height will be minimized when all \( n \) nodes are packed as tightly as possible, i.e. when the proper binary tree is also a full binary tree. In a full binary tree, of height \( h \), the total number of nodes is:
\[
2^0 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1, \ i.e., \ h = \log_2(n+1) - 1. \ If \ the \ tree \ T \ is \ not \ full, \ the \ height \ h \ will \ only \ increase. \ We \ can \ thus \ conclude \ that \ h \geq \log_2(n+1) - 1, \ for \ any \ proper \ binary \ tree \ T \ having \ n \ nodes. \ □

4. Consider the binary tree \( T \) in Figure (1). Write down the order of the nodes, when you traverse the tree in inorder, preorder and postorder. (6 points)

![Binary Tree T](image)

Figure 1: Binary Tree \( T \)

Observe that in an inorder traversal, the left children of a node are visited before it is visited and the right children of a node are visited after it is visited. Applying this recursively, we conclude that the nodes in \( T \) would be visited in the following order: \(-1, 1, 8, 2, 3, 4, 5, 9, 6, 7\).
Observe that in a preorder traversal, a node is visited before its children are visited and the left children of a node are visited before the right children are visited. Applying this recursively, we conclude that the nodes in T would be visited in the following order: 4, 2, 1, −1, 8, 3, 5, 6, 9, 7.

Observe that in a postorder traversal, a node is visited after its children are visited and the left children of a node are visited before its right children are visited. Applying this recursively, we conclude that the nodes in T would be visited in the following order: −1, 8, 1, 3, 2, 9, 7, 6, 5, 4.

5. Prove that Algorithm (0.1) correctly sorts an n-input sequence S provided as an n-element array A (in increasing order). You may assume that the n elements of the array are stored in the locations A[1], A[2], ..., A[n]. What is the worst-case running time of the algorithm? (8 points)

Hint: You may either use the Loop Invariant Technique or induction (second principle!) on the number of elements in the array!

<table>
<thead>
<tr>
<th>Function Bubble-Sort(A, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: for (i = 1 to n − 1) do</td>
</tr>
<tr>
<td>2: for (j = i + 1 to n) do</td>
</tr>
<tr>
<td>3: if (A[i] &gt; A[j]) then</td>
</tr>
<tr>
<td>4: temp = A[i]</td>
</tr>
<tr>
<td>6: A[j] = temp</td>
</tr>
<tr>
<td>7: end if</td>
</tr>
<tr>
<td>8: end for</td>
</tr>
<tr>
<td>9: end for</td>
</tr>
</tbody>
</table>

*Algorithm 0.1: Bubble Sort Algorithm*

**Proof:** We shall discuss correctness of the Bubble-Sort() Algorithm using the Loop invariant technique (Please see Pg. 27 of [GT02]).

We use the following loop invariant:

S(i): The first i − 1 elements are in their correct positions in A.

The key difference between our approach and the approach in [GT02], is that we start from S(1) since our elements are stored in A[1], A[2], ..., A[n] as opposed to A[0], A[1], ..., A[n − 1].

S(1) is trivially true, since A[0] does not exist. Consider the working of the outer loop in iteration i = k. Prior to the start of this iteration, we have A[1] ≤ A[2] ≤ ... ≤ A[k − 1], with A[k − 1] being the (k − 1)th smallest element in A. As iteration i = k proceeds, we scan through the array to determine the smallest element in A[k] through A[n] and put it in A[k]. Hence, if S(1), ..., S(k) are true, then S(k + 1) is true, i.e., after the i = k iteration (and before the i = k + 1 iteration), we have A[1] ≤ A[2] ≤ ... ≤ A[k − 1] ≤ A[k] and A[k] is the kth smallest element in A. It follows that S(n) is true, i.e., at the end of the iteration i = n − 1, the first n − 1 elements are in their correct positions in A. This forces A[n] to be in its correct place!

Thus, we have shown that the algorithm is correct by applying the principle of loop invariants.

A rough approximation to the running time can be obtained by observing that the i loop runs at most n times and so does the j loop. Further, within the nested for loops, at most 4 statements are executed. So the total running time cannot exceed 4 · n², i.e., O(n²). We give a more formal analysis below. Let T(n) denote the worst-case running time of Algorithm (0.1). We then have

\[
T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 4 = 4 \sum_{i=1}^{n-1} (n - i)
\]
\[
= 4 \cdot \left( \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \right) \\
= 4 \cdot \left( n \cdot \sum_{i=1}^{n-1} 1 - \frac{n \cdot (n-1)}{2} \right) \\
= 4 \cdot \left( n \cdot (n-1) - \frac{n \cdot (n-1)}{2} \right) \\
= 4 \cdot \frac{n \cdot (n-1)}{2} \\
= O(n^2)
\]

In passing, we note that there is no good input for this algorithm. The if statement within the double for loop is executed \( \Omega(n^2) \) times. \( \Box \)

References