Fundamental Algorithms 3

– Solution Examples –

Exercise 1

Consider a partitioning algorithm that, in the worst case, will partition an array of \( m \) elements into two partitions of size \( \lfloor \epsilon m \rfloor \) and \( \lceil (1 - \epsilon) m \rceil \), where \( \epsilon \) is fixed, and \( 0 < \epsilon < 1 \). Show that a quicksort algorithm based on this partitioning has a worst-case complexity of \( O(n \log n) \).

Solution:

Again, we will only count comparisons between array elements.

Using that the partitioning step will require at most \( n \) comparisons, we get the following recurrence for the necessary number \( C(n) \) of comparisons:

\[
C(1) = 0
\]

\[
C(n) = C(\epsilon n) + C((1 - \epsilon) n) + n
\]

We guess \( C(n) := an \log_2 n + b \) as the solution, and try to find constants \( a \) and \( b \) such that the recurrence is satisfied:

**case** \( n = 1 \):

\[
C(1) = a \cdot 1 \log_2 1 + b = 0 \;
\Rightarrow b = 0,
\]

hence, \( C(n) = an \log_2 n \).

**case** \( n > 1 \): We insert our guess into the recurrence:

\[
an \log_2 n = C(n) = C(\epsilon n) + C((1 - \epsilon) n) + n
\]

\[
\Leftrightarrow an \log_2 n = a\epsilon n \log_2 (\epsilon n) + a(1 - \epsilon) n \log_2 ((1 - \epsilon) n) + n
\]

\[
\Leftrightarrow an \log_2 n = a\epsilon n (\log_2 \epsilon + \log_2 n) + a(1 - \epsilon) n (\log_2 (1 - \epsilon) + \log_2 n) + n
\]

\[
\Leftrightarrow an \log_2 n = a\epsilon n \log_2 \epsilon + a\epsilon n \log_2 n +
\]

\[
a(1 - \epsilon) n \log_2 (1 - \epsilon) + a(1 - \epsilon) n \log_2 n + n
\]

\[
\Leftrightarrow an \log_2 n = a\epsilon n \log_2 \epsilon + a\epsilon n \log_2 n +
\]
Thus, the recurrence is satisfied if

\[ C(n) = \frac{-n \log_2 n}{\varepsilon \log_2 \varepsilon + (1-\varepsilon) \log_2 (1-\varepsilon)} \]

Note that the constant \( a \) will be very large for values of \( \varepsilon \) that are close to either 0 or 1. Thus, even very bad partitions will not destroy the \( \mathcal{O}(n \log n) \) complexity, provided that the respective partition sizes are bounded by \( \varepsilon n \) and \( (1-\varepsilon) n \). However, bad partitions will still lead to slow algorithms due to the large constant factor involved.

K-Exercise 2 (An Iterative MergeSort)

The following iterative implementation of the MergeSort algorithm is proposed:

\begin{verbatim}
ItMergeSort (A: Array [0..n-1]) {
  // n assumed to be a power of 2: n=2^k
  k := log2 (n)
  //
  m := 2
  for L from 1 to k do {
    for i from 0 to (n/m)-1 do {
      MergeIP (A[ i*m .. i*m+(m/2-1) ,
                A[ i*m+(m/2) .. i*m+(m-1) ];
      }
    m := 2*m;
  }
}
\end{verbatim}

The procedure MergeIP is equivalent to the procedure Merge discussed in the lecture, but can work directly on the array A (i.e., merges two adjacent subarrays of A).

a) Describe shortly and in plain words, how ItMergeSort compares to the recursive MergeSort implementation discussed in the lecture. For that purpose, draw a diagram that illustrates the sorting of an array A[0..7] for ItMergeSort.

b) Formulate a loop invariant for the L-loop of the algorithm, and prove its correctness.

Solution:

a) In each iteration of the L-loop two adjacent subarrays are merged. The lengths of the merged subarrays \( (m/2) \) is doubled from each L-loop iteration to the next. In that way, the same
merging steps as for the recursive implementation of MergeSort are executed. The divide 
steps are implicitly performed on the array.

|-------|------|------|------|------|------|------|------|------|

|-------|------|------|------|------|------|------|------|------|

|-------|------|------|------|------|------|------|------|------|

b) We propose the following loop invariant:

At entry of the L-loop, the array $A$ consists of $\frac{2^n}{m}$ subarrays of length $\frac{m}{2}$, where $m = 2^L$. Each of the subarrays is sorted.

Here’s a sketch of the proof:

**Initialisation:** on the first entry, for $L = 1$ and $m = 2^1$, the length of the subarrays is claimed to be $\frac{m}{2} = 1$ with $\frac{2^n}{2} = n$ subarrays – this is obviously satisfied, as subarrays of length 1 are always sorted.

**Maintenance:** The i-loop will take $\frac{n}{m}$ pairs of two adjacent subarrays and merge them using the procedure MergeIP. Provided the correctness of MergeIP, this will lead to $\frac{n}{m}$ subarrays of twice the length, which satisfies the loop invariant for the next iteration. Note that $m$ is multiplied by 2, to retain $m = 2^L$.

**Termination:** At termination, $L = k + 1$ and thus $m = 2^{k+1} = 2n$. Hence, we have only $\frac{2^n}{2n} = 1$ subarray of length $\frac{2^n}{2} = n$, which is sorted. This implies the correctness of the sorting algorithm.