CPS 130 Homework Solutions

Guangwei Yuan
Fall 1999

HOMEWORK 3

3.1: (a) The recurrence equation of the running time for merge sort is: $2T(n/2) + \Theta(n)$, which solves to be $\Theta(n \log n) = \Omega(n \log n)$ for all cases.

(b) The procedure `IMPROVED-MERGE-SORT(A, p, r)` sorts the elements in the subarray $A[p..r]$, given $p < r$.

```python
IMPROVED-MERGE-SORT(A, p, r)
    if A[p..r] is already sorted
        then return
    else $q \leftarrow \lceil (p+r)/2 \rceil$
        IMPROVED-MERGE-SORT(A, p, q)
        IMPROVED-MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

That is, in each recursive step, we improve the original merge sort by first checking the subarray to see if it is already sorted. This checking takes $O(n)$ time.

(c) If $k = 1$, then the recurrence equation of the running time for the improved merge sort turns to be: $T(n/2) + O(n) + O(n) = T(n/2) + O(n)$, because one of the two subarrays is already sorted, and there will be no further operations on it according to the algorithm. The running time is therefore $O(n)$.

(d) To extend to arbitrary $k$, we use induction proof on $k$. Here we have the recurrence: $T(n, k, i) = T(n/2, l(i), i + 1) + T(n/2, k - l(i), i + 1) + O(n)$. Here, $i$ indicates the level of the recurrence, with its initial value being 0. $l$ is the insertion index of one of the two subarrays, and $l$ is a function of $i$. Then the insertion index of the other subarray is $k - l$. We want to prove that $T(n) = O(\min(nk, n \log n))$.

(i) For the base case when $k = 1$, $T(n) = O(n) = O(\min(nk, n \log n))$.

(ii) Suppose for insertion index less than $k$, it holds.
\[(iii) \quad T(n, k, 0) = T(n/2, l(0), 1) + T(n/2, k - l(0), 1) + O(n)\]
\[= O\left(\min\left(\frac{n^k}{2}, \frac{n}{2} \log \frac{n}{2}\right)\right) + O\left(\min\left(\frac{n(k - l(0))}{2}, \frac{n}{2} \log \frac{n}{2}\right)\right) + O(n)\]
\[\leq O\left(\min\left(\frac{nk}{2}, \frac{n}{2} \log n\right)\right) + O\left(\min\left(\frac{nk}{2}, \frac{n}{2} \log n\right)\right) + O(n)\]
\[= O\left(\min(nk, n \log n)\right) + O(n)\]
\[= O\left(\min(nk, n \log n)\right)\]

3.4: (a)

\textbf{RANDTRIT}
\begin{itemize}
  \item \texttt{x0} $\leftarrow$ \texttt{RANDBit()} \\
  \texttt{x1} $\leftarrow$ \texttt{RANDBit()} \\
  \texttt{x} $\leftarrow$ (\texttt{x1} $\ll$ 1) + \texttt{x0} \\
  \textbf{if} \texttt{x} < 3 \\
  \textbf{then} \textbf{return} \texttt{x} \\
  \textbf{else} \textbf{RANDTRIT}()
\end{itemize}

(b) The expected number of calls to \texttt{RANDBit} is: $2 \sum_{i=0}^{\infty} \frac{1}{4^i} = \frac{8}{3}$

(c) $2n$ with the probability of $\frac{1}{4^{n-1}}$. So its worst case bound is infinite.

3.5: To construct a bad example, we can let the algorithm itself do the work for us. Every time the algorithm examines one (or several) elements to select a pivot, we set those array elements to be the smallest element not yet used as a pivot. For instance, we begin by setting all elements examined in determining the first pivot to 0. This means that the first pivot selected will be 0, and all non-examined elements will end up in the GREATER pile (assuming they are greater than 0 which they will be by construction) On the next round, we set all elements examined to 1, and so on. When we are done simulating the sorting algorithm to completion, we can work backwards to reconstruct what this array would have looked like at the start.

In each iteration of the algorithm, all but some constant $c$ elements go into the GREATER pile. This generates a recurrence of $T(n) = T(n-c) + \Theta(n)$. This recurrence solves to $\Theta(n^2)$, which is $\Omega(n)$ as desired.