CS140: Algorithms

Spring 2001

Solutions Assignment 5

1. $S = S[0], S[1], \ldots, S[n-1]$ is an array of distinct integers. S is said to be *unimodal* if there is some index i (where $0 \leq i \leq n-1$) such that $S[0], ..., S[i]$ is a strictly increasing sequence and $S[i], ..., S[n-1]$ is a strictly decreasing sequence. For example, $S = 2, 4, 8, 7, 6, 5, 3, 1, 0, -1$ is unimodal but $S = 2, 3, 1, 4$ is not.

   Consider the problem of finding the maximum element in a unimodal array S when $n = 2^m$. Assume an algorithm is only allowed access to S through queries of the form "Is $S[j] < S[k]$?" Give the best upper and lower bounds you can on the number of queries needed to solve this problem. Your answer should not use big-O notation; i.e. you can get exact bounds for this problem. Prove your answers; i.e. give an algorithm that achieves the upper bound and an adversary that achieves the lower bound. Exactly m queries are necessary and sufficient (in worst case) to determine the index of the max element. To prove m queries are sufficient consider the following algorithm.

   `algorithm` Unimodal($n, S[1], \ldots, S[n]$)

   ```
   if n==1 return 1
   if S[n/2] < S[n/2 +1]
      return n/2 + Unimodal(n/2,S[n/2 +1], ..., S[n])
   else return Unimodal(n/2,S[1], ..., S[n/2])
   ```

   `end algorithm`

   If $n = 1$ the algorithm returns the correct index. Now suppose that $n = 2^m$, where $m > 1$. Assume i is the index of the maximum element. If $i \leq n/2$ then $S[n/2]$ must be greater than $S[n/2+1]$. In this case the algorithm recurses on the unimodel set $S[1], \ldots, S[n/2]$. We can assume the recursive call returns the index of the maximum element of $S[1], \ldots, S[n/2]$, which must be i. Thus the algorithm returns the correct index for $S[1], \ldots, S[n]$.

   If $i > n/2$ then $S[n/2] < S[n/2+1]$ and the algorithm recurses on the set $S[n/2+1], \ldots, S[n]$. The $i - n/2$ element in this set (we assume the elements are renumbered starting at 1 in the recursive call) is the maximum element and we can assume the recursive call returns this value. But the algorithm then returns i as the index of the maximum element in the set $S[1], \ldots, S[n]$. Therefore the algorithm answers correctly.

The number of queries made by the algorithm is $T(n) = 0$ if $n = 1$ and $T(n) = T(n/2) + 1$ if $n = 2^m$ for $m > 0$. This recurrence is satisfied by $T(n) = \lg(n) = m$.

To prove a lower bound we'll construct an adversary. First consider how much information the algorithm gains from a query "Is $S[j] < S[k]$?" (where $j < k$). If the answer is yes the algorithm infers that the index of the maximum element is larger than j. If the answer is no the algorithm infers that the maximum element has an index smaller than k.

The adversary maintains a range of indices $[lo, hi]$ that are possible indices of the maximum element. Initially $lo = 1$ and $hi = n$. If the algorithm asks "Is $S[j] < S[k]$?" (where $lo \leq j < k \leq hi$) the adversary answers yes if $hi - j \geq k - lo$ and no otherwise. Then the algorithm can (at best) halve the range of possible indices with each query. Further, the algorithm cannot halt until $lo = hi$. Thus the algorithm must make $lg(n)$ queries before halting.

2. In class we showed that any comparison-based sorting algorithm is $\Omega(n \lg n)$. This bound does not necessarily hold if we restrict the input in some way. For example, the input may be "semi-sorted" to begin with. Consider the following definitions of "semi-sorted" and determine whether the problem of sorting an input that satisfies the definition can break the $\Omega(n \lg n)$ bound. Prove your answer.

   (a) The input array is a min-heap.

   The $\Omega(n \lg n)$ bound still holds. To see this notice that we can build a heap for arbitrary input of size n in $O(n)$. Thus if we could sort the heap in $o(n \lg n)$ time we would have a comparison-based sorting algorithm that is $o(n \lg n)$, which is impossible.

   (b) Each element in the input array is within 5 positions of its sorted position. To be more precise, the rank of element s_i in the input list is between $i - 5$ and $i + 5$. (Assume the input integers are distinct.)

   The following algorithm sorts this type of input in $O(n)$ time.

   `algorithm` SortAlmostSorted($S = \{s_1, \ldots, s_n\}$)

       `for` $i = 1$ `to` $n - 5$ `do`

           `InsertionSort`($\{s_i, s_{i+1}, \ldots, s_{i+5}\}$)

       `end`

   `end algorithm`