CS140: Algorithms

Fall 2000

Solutions to Assignment 4

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. The input in this problem is a binary vector $b_0, \ldots, b_{n-1}$. The problem is to determine whether the vector contains two consecutive 0's. You are allowed to make queries of the form "What is bit i?" Let $T(n)$ denote the number of queries needed to solve the problem on a vector of n bits. Give the best upper and lower bounds you can on $T(n)$. Prove your answers; i.e. give an algorithm that achieves the upper bound and an adversary that achieves the lower bound.

   Hint: Consider finding a bound for the following as well. Let $S(n)$ denote the number of queries needed to determine whether a bit vector of length n satisfies (at least) one of the following conditions:

   (a) It contains two consecutive 0's

   (b) The first bit is a 0.

   Define $f(n) = n - 1$ if $n \equiv 1 \bmod 3$ and $f(n) = n$ otherwise. Claim: T(n)=f(n).

   First we prove the upper bound; i.e $T(n) \leq f(n)$. We'll design an algorithm that makes at most $f(n)$ queries in solving the problem. As shown in class, there is an algorithm that achieves the bound for $n = 0, 1, 2, 3$. Assume the an algorithm achieves the bound for input size less than n, where $n > 3$. We'll show how to extend the algorithm for inputs of size n.

   If $b_1 = 1$

   Recurse on bits $b_2 \ldots b_{n-1}$.

   Else

   Query bits b_0 and b_1. If either are 0 return yes. If not, recurse on bits $b_3 \ldots b_{n-1}$.

If $b_1 = 1$ the algorithm makes $T(n) = 1 + T(n-2) \leq n - 1 \leq f(n)$ queries. If $b_1 = 0$ the algorithm makes $T(n) = 3 + T(n-3) = f(n)$. Thus $T(n) \leq f(n)$.

Now we prove that this is also a lower bound; i.e. $T(n) \geq f(n)$. Again, we showed in class that an adversary can force any algorithm to make $f(n)$ queries for $n = 0, 1, 2, 3$. Assume the adversary achieves the bound for input size less than n, where $n > 3$. We'll show how to extend the adversary for inputs of size n.

The proof will follow from two observations. First, the adversary never admits to double zeroes. Second, for any i and n such that $0 \leq i \leq n-1$ one of the following holds:

(a) Neither i nor $n - i - 1$ are equivalent 1 modulo 3.

(b) Neither $i - 1$ nor $n - i - 2$ are equivalent 1 modulo 3.

(c) n and at most one of $i - 1$ and $n - i - 2$ are equivalent 1 modulo 3.

Now we describe the adversary. Suppose the algorithm's first query is b_i, $0 \leq i \leq n - 1$.

If neither i nor $n - i - 1$ are equivalent 1 mod 3 the adversary will say $b_i = 1$. The adversary then recurses to answer queries on the two subproblems $b_0 \ldots b_{i-1}$ and $b_{i+1} \ldots b_{n-1}$.

Else the adversary answers $b_i = 0$. Further the adversary decides that b_{i-1} (when $i - 1 \geq 0$) and b_{i+1} (when $i - 1 \leq n - 1$) are both 1. The adversary then recurses to answer queries on the two subproblems $b_0 \ldots b_{i-2}$ and $b_{i+2} \ldots b_{n-1}$.

It is clear that the adversary never admits to double zero's. Thus any algorithm must solve both subproblems following its query. If the adversary answers $b_i = 1$ the algorithm must ask $1 + T(i) + T(n - i - 1) \geq f(n)$ queries, since neither i nor $n - i$ are equivalent to 1 modulo 3. If the adversary answers $b_i = 0$ then the algorithm must, at some point, query b_{i-1} and b_{i+1} to confirm they are non-zero. Thus the algorithm must ask $3 + T(i - 2) + T(n - i - 2)$ queries. By our second observation this is at least $f(n)$. (The boundary cases follow by the same arguments.)