CSE101 - HW #2

Sample Solution

1 Problem Description

Suppose you have an array of integers $a[k]$, $k = 1, \ldots, n$. You know that there exists an index $i \in \{1, \ldots, n\}$ such that

$$a[k] < a[k+1] \quad \text{for} \quad k < i$$

and

$$a[k] > a[k+1] \quad \text{for} \quad k > i.$$  

This means the sequence is first strictly increasing, then strictly decreasing, and $a[i]$ is the biggest element. Note: Given are the number $n$ and the numbers $a[k]$, $k = 1, \ldots, n$. It is not excluded that $i = 1$ [or $i = n$], which means the whole sequence is strictly decreasing [increasing].

a. Give an efficient algorithm that finds the largest element, i.e. determines $i$. Give a worst case time analysis (big $\mathcal{O}$ notation) in terms of $n$.

b. For a given number $b$, give an efficient algorithm that searches the array for $b$ and returns an index $k$ such that $a[k] = b$ or $k = 0$ for ”$b$ is not in the array”. Give a worst case time analysis (big $\mathcal{O}$ notation) in terms of $n$.

2 Sample Solution

a. General idea. Let $i$ denote the index of the biggest element in the list. Note that if we take any index $k$, we can easily find out if $i \leq k$ or $i > k$ by checking if $a[k] < a[k+1]$:  

$$a[k] < a[k+1] \quad \Rightarrow \quad i > k$$

$$a[k] > a[k+1] \quad \Rightarrow \quad i \leq k$$

Now we can use an approach similar to binary search to find $i$: Pick $k$ to be an index approximately in the middle of the input array. Then find out if the “peak” is left of $k$ or $k$ itself by checking $a[k]$ and $a[k+1]$. If so, proceed by calling the algorithm recursively on $a[1], \ldots, a[k]$. If not, proceed with $a[k+1], \ldots, a[n]$.

Algorithm. The given algorithm expects that the size of the array is stored in $n$, the elements in $a[1], \ldots, a[n]$ when started. Upon termination, it returns the index of the biggest element. Furthermore, the program maintains two pointers
lower and upper, which are used to store the lower and upper bound on \( i \) in the actual iteration. In other words: During all the time, we know that
\[
i \in \{lower, \ldots, upper\}
\]
When lower becomes equal to upper, it is \( i = lower = upper \). Note that \( upper - lower \) decreases in each iteration by at least \( \lfloor \frac{upper - lower}{2} \rfloor \) (do you see why?), which is always bigger or equal to 1 (unless upper = lower, which means we are done). Therefore we are guaranteed that the program stops in \( \Theta(\log n) \) steps. This can also be seen by considering the fact that this algorithm behaves almost exactly like binary search, except for that it needs to look at two elements per iteration.
Note also that this algorithm never tries to access an array element out of bounds, like \( a[0] \) or \( a[n + 1] \). (Why?)

1. program FindMax(a, lower, upper)
2. while (lower < upper) do // while not found
3. \[ k := \lfloor \frac{upper + lower}{2} \rfloor \] // integer division, take approx. the middle
4. if (a[k] < a[k + 1])
5. \[ lower := k + 1; \] // \( i \) must be strictly bigger than \( k \)
6. else
7. \[ upper := k; \] // \( i \) must be \( \leq k \)
8. end while;
9. return lower; // index found, return

b. General idea. Use a) to find the index \( i \). Then use binary search on both "sides" of \( i \).

Algorithm. I just give a high-level description of the algorithm here, as the "sub"-algorithm binary search is discussed in detail in Chapter 2.3 (p. 105) in the textbook. The version of binary search that searches a decreasing list differs from Algorithm 2.5 on page 106 just by exchanging the < and > signs. As we make 3 calls to algorithms that run in \( \Theta(\log n) \), this runs also in \( \Theta(\log n) \).

1. program FindElement(a, lower, upper, b)
2. \( i = \text{FindMax}(a, \text{lower, upper}); \)
3. \( k = \text{BinarySearch}(a, \text{lower, } i, b); \)
4. if \( k = 0 \)
5. \( k = \text{BinarySearchDecreasing}(a, i, \text{upper, } b); \)
6. return \( k \);