COP4531 – Complexity and Analysis of Data Structures and Algorithms **Fall 2017**

Homework 2: Deadline Friday 10/06/2017

Instructor: Viet Tung Hoang

This homework consists of two parts: (1) Implementation, and (2) Regular problem set. For the implementation, please email your files to the TA Vigneshwar Padmanaban (vp15g@my.fsu.edu). If you have specific programming questions on the implementation, Vignesh will also be the person to ask. For the regular problem set, if for some reason you can't show up in class to submit the solution then please email it to the grader Xiaoxiao Gan (xg16c@my.fsu.edu). For the problem set, you don't need to write pseudocode; English description of your algorithm will be fine.

1. (20 points) Let A be an array of n (possibly negative) **integers** such that $A[1] < A[2] < \cdots < A[n]$. Find an $O(\log(n))$ divide-and-conquer algorithm which either finds an $i \in \{1, \ldots, n\}$ such that $A[i] = i$, or else determines that there is no such i. Justify the correctness and running time of your algorithm. (Hint: The solution is very similar to binary search. Moreover, you also need to make use of the fact that the elements of this array are integers, and thus for every $1 \le j < k \le n$, we have $A[j] \le A[k] - (k - j)$.)

Solution: We will loosely follow the structure of binary search as follows. For the base case $n = 1$, we can simply check if $A[1] = 1$. For $n > 1$, handle the problem recursively as follows. Let $mid = \lceil n/2 \rceil$. If $A[mid] = mid$ then we are done. If $A[mid] > mid$ then for any $k > mid$, we have

$$A[k] \ge A[mid] + (k - mid) > mid + (k - mid) = k,$$

and thus we only need to continue searching in the left subarray $A[1 : mid - 1]$. Likewise, if $A[mid] < mid$ then for any $j < mid$, we have

$$A[j] \le A[mid] - (mid - j) < mid - (mid - j) = j$$

and thus we only need to continue searching in the right subarray $A[mid + 1 : n]$. This algorithm has the same running time as binary search, and thus it takes only $O(\log(n))$ time.

Formally, below is the pseudocode that describes procedure Search(A, min, max) that searches the array A from index min to index max (inclusive). If it can't find a solution then Search returns -1. Thus one should run Search$(A, 1, n)$ to search the entire array A.

```
Search($A, min, max$)
// Base case
If $min < max$ then return $-1$
If $min = max$ then
    If $A[min] = min$ then return $min$ else return $-1$
// Recursive case
$mid \leftarrow \lfloor (min + max)/2 \rfloor$
If $A[mid] = mid$ then return $mid$
Else if $A[mid] > mid$ then return Search($A, min, mid - 1$)
Else return Search($A, mid + 1, max$)
```

2. (20 points) Exercise 3 in page 246–247, Chapter 5 of the textbook.

Solution: Let $e_1, \ldots, e_n$ denote the equivalence classes of the cards: cards i and j are equivalent if $e_i = e_j$. What we are looking for is a value x so that more than $n/2$ of the indices have $e_i = x$.

Divide the set of cards into two roughly equal piles: a set of $\lfloor n/2 \rfloor$ cards and a second set for the remaining $\lceil n/2 \rceil$ cards. We will recursively run the algorithm on the two sides, and will assume that if the algorithm finds an equivalence class containing more than half of the cards, then it returns a sample card in the equivalence class.

Note that if there are more than $n/2$ cards that are equivalent in the whole set, say have equivalence class x, then at least one of the two sides will have more than half the cards also equivalent to x. So at least one of the two recursive calls will return a card that has equivalence class x.

The reverse of this statement is not true: there can be a majority of equivalent cards in one side, without that equivalence class having more than $n/2$ cards overall (as it was only a majority on one side). So if a majority card is returned on either side we must test this card against all other cards.

The correctness of the algorithm follows from the observation above: that if there is a majority equivalence class, then this must be a majority equivalence class for at least one of the two sides.

To analyze the running time, let $T(n)$ denote the maximum number of tests the algorithm does for any set of n cards. The algorithm has two recursive calls, and does at most $2n$ tests outside of the recursive calls. So we get the following recurrence (assuming n is divisible by 2):

$$T(n) \leq 2T(n/2) + 2n \quad .$$

Thus $T(n) = O(n \log(n))$.