CS21003 - Tutorial 5

Solution Sketch

1. Let A be a sorted array of n distinct elements. An array B is formed by cyclically right-shifting the array A by some k cells. Given B (but not A or k), determine how the shift amount k can be computed in $O(logn)$ time.

   ⇒ Suppose we shift the array $[2, 5, 8, 13, 15, 19]$ by $k = 2$ cells, we get an array $[15, 19, 2, 5, 8, 13]$. We see that the portion of B from 0 to $k - 1$ is an increasing sequence, and so is the portion between k to $n - 1$. Each entry in the first sequence is larger than the second.

```
findShift(A[],n)
L=0, R = n-1
while(true){
M=(L+R)/2
//Check if M is at the boundary
if ((A[M-1]<A[M]) && (A[M]>A[M+1])) return M+1
if ((A[M-1]>A[M]) && (A[M]<A[M+1])) return M

if (A[M] > A[0]) L = M + 1; /* M is inside the first sequence */
else R = M - 1; /* M is inside the second sequence */
}
```

2. Suppose you are given two *sorted* arrays A and B with n and m integer elements each. Provide an $O(log(max(m, n)))$ algorithm to find the median of these $n + m$ elements.

   ⇒ The median of the two arrays should be greater than $x = floor((m + n)/2)$ elements. The median will be either in A or in B. We systematically do binary search in both A and B.

   Suppose we are searching in A. Rough idea below:

```
L=0, R=n-1
M=(L+R)/2
//For A[M] to be the median, it should be greater than x elements. It
is already greater than M elements in A. Now compare A[M] with
B[x-M].

If A[M] = B[x-M] return A[M]
else if A[M] > B[x-M]//need to search left
R=M-1
else L=M+1
```

   Complexity - $O(logm + logn) = O(log(max(m, n)))$.

3. Assume you have an array $A[1,\dots,n]$ of n elements. A majority element of A is any element occurring in more than $n/2$ positions. Assume that elements cannot be ordered or sorted, but can be compared for equality. Design a divide and conquer algorithm to find a majority element in A (or determine that no majority element exists) to run in $O(nlogn)$ time.

   $\Rightarrow$ The algorithm begins by splitting the array in half repeatedly and calling itself on each half. When we get down to single elements, that single element is returned as the majority of its (1-element) array. At every other level, it will get return values from its two recursive calls.

   The key to this algorithm is the fact that if there is a majority element in the combined array, then that element must be the majority element in either the left half of the array, or in the right half of the array (or both). There are 4 scenarios.

   a. Both return "no majority. Then neither half of the array has a majority element, and the combined array cannot have a majority element. Therefore, the call returns "no majority.

   b. The right side is a majority, and the left isnt. The only possible majority for this level is with the value that formed a majority on the right half, therefore, just compare every element in the combined array and count the number of elements that are equal to this value. If it is a majority element then return that element, else return "no majority.

   c. Same as above, but with the left returning a majority, and the right returning "no majority.

   d. Both sub-calls return a majority element. Count the number of elements equal to both of the candidates for majority element. If either is a majority element in the combined array, then return it. Otherwise, return "no majority. The top level simply returns either a majority element or that no majority element exists in the same way.

   Running time: $\Theta(nlogn)$

4. The maximum partial sum problem (MPS) is defined as follows. Given an array $A[1,\dots,n]$ of integers, find values of i and j with $1 \le i \le j \le n$ such that $\sum_{k=i}^{j} A[k]$ is maximized. Example: For the array $[4,-5,6,7,8,-10,5]$, the solution to MPS is $i = 3$ and $j = 5$ (sum 21). Can you think of a brute force solution in $O(n^2)$? How can you use divide and conquer to improve this to $O(nlogn)$?

   $\Rightarrow$ Divide and conquer divides the array into two equal parts. The sequence can either entirely lie in any of the two parts or should cross. We can take the maximum of the solutions for the both the parts, as well as the crossing sequence to combine.

   If we can find the sum of the crossing sequence in $O(n)$ time, we have a solution in $O(nlogn)$ time. This goes as follows:

   For the left sequence, compute the maximum sum (including elements) backwards. For the right sequence, compute the maximum sum (including elements till that point) forward. Add these two. For instance if [4,-5,6,7] and [8,-10,5] are two parts, going backwards in left: [7,13,8,12] – max is 13 for [6,7]. Going forward in right: [8,-2,3] – max is 8 for [8]. So, the crossing sequence with the highest sum is [6,7,8].

5. We discussed in class the algorithm to solve the selection problem in $O(n)$ time. The algorithm takes group of 5 elements. What happens if you take groups of 3 or 7 instead?

   $\Rightarrow$ Try it yourself and verify that a group of 5 elements is the optimal choice.

6. Can you generalize your algorithm for the question 2 for $k-th$ order statistics of the two arrays combined, and improve the algorithm to work in $O(log(min(m,n)))$?

   $\Rightarrow$ Generalizing the algorithm for k th order statistics is easy. Replace x by $k-1$. To imrpove the algorithm to work in $O(log(min(m,n)))$ – Let the two arrays be of size 100 and 10. Convince yourself that if the median lies in A (size 100), it would suffice to search between index 45 to 55. This take the complexity to $O(log(min(m,n)))$.

7. To solve problem 3, you can also use a divide and conquer strategy to work in $O(n)$ time. Describe the algorithm. **Hint:** Assume that n is even. Arbitrarily pair elements of A. In each pair (a_i, a_j), check whether $a_i = a_j$. If so, write this element in an array B. If not, discard the pair. The array B consists of at most $n/2$ elements after all pairs are considered. Recursively compute the majority of B. How can you use this to find the final solution?

   $\Rightarrow$ Let a be the majority element in A. Suppose that a is paired with itself n_1 times and with other elements n_2 times. Also let n_3 be the number of pairs not involving a as either member. Then, B contains at most $n_1 + n_3$ elements. Since a is a majority element in A, we have $2n_1 + n_2 > n/2$. Moreover, we have $2(n_1 + n_2 + n_3) = n$. Therefore, $4n_1 + 2n_2 = 4n_1 + (n2n_12n_3) = 2n_12n_3 + n > n$, that is, $n_1 > n_3$, that is, a is the majority element in B too. However, the converse of this is not true. For example, let A consist of the pairs (a, a) and (b, c). It follows that if the recursive call returns "no majority element in B, then A too does not contain a majority element. However, if the recursive call returns a majority element a, we need to check whether a is actually the majority element of A. In any case, the running time of this recursive algorithm satisfies $T(n) \leq T(n/2) + \Theta(n)$, that is, $T(n) = O(n)$.