CS 4349.400 Homework R

Due Wednesday October 24th, in class

This is an extra credit assignment worth the same amount as one normal homework. You are under no obligation to complete this homework if you are satisfied with your midterm grade, and it will not be used as your one dropped homework should you choose to skip it.

This assignment is meant to give you another chance to go over material from the first half of the class. The questions are meant to guide you through designing and analyzing a few algorithms, and you may find them easier than the ones given earlier in the semester.

1. Consider the following generalization of the Blum-Floyd-Pratt-Rivest-Tarjan SELECT algorithm we discussed in class. It partitions the input array into $\lceil n/7 \rceil$ blocks of size 7 instead of $\lceil n/5 \rceil$ blocks of size 5 but is otherwise identical. In the pseudocode below, the necessary modifications are indicated in red.

```
MOM7SELECT(A[1 .. n], k):
  if n ≤ 49
      use brute force
  else
      m ← ⌈n/7⌉
      for i ← 1 to m
          M[i] ← MEDIANOF7(A[7(i − 1) + 1 .. 7i])
      mom7 ← MOM7SELECT(M[1 .. m], ⌊m/2⌋)

      r ← PARTITION(A[1..n], mom7)

      if k < r
          return MOM7SELECT(A[1 .. r − 1], k)
      else if k > r
          return MOM7SELECT(A[r + 1 .. n], k − r)
      else
          return mom7
```

   Our goal is to analyze the running time of this algorithm.

   (a) In the original algorithm (with blocks of size 5), we observed that $3n/10$ elements were smaller than the median-of-medians. How many elements are smaller than the median-of-medians when we use blocks of size 7?

   (b) Using the previous observation, and the fact that the algorithm takes linear time outside the recursion calls, we observed the worst-case running time of the original selection algorithm to obey the recurrence $T(n) = T(n/5) + T(7n/10) + n$. State a recurrence for the running time of MOM7SELECT.

   (c) To solve the running time recurrence, we need to sum over all node values in the recursion tree. What is the sum of the nodes values on the ith level of the tree (the root is at level 0)?

(d) Finally, express the asymptotic solution to your recurrence using big-O notation and justify your answer. If your previous answers are correct, then the solution to your recurrence should be $O(n)$.

2. An ***inversion*** in an array $A[1 .. n]$ is a pair of indices i, j such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward). Our goal is design a divide-and-conquer algorithm to count the inversions in an n-element array in $O(n \log n)$ time. You may want to read through each part of this question before you begin.

   (a) Suppose you had access to a procedure CountCrossInversions($A[1 .. n], m$) that, in $O(n)$ time, counts the number of inversions i, j in A where $1 \le i \le m$ and $m+1 \le j \le n$. Describe a divide-and-conquer algorithm for counting inversions that uses one call to CountCrossInversions and two recursive calls to itself. You should justify the correctness of your algorithm, but do not worry about running time for this part.

   For this part, you must use the procedure CountCrossInversions to access A. You may not compare members of A directly.

   (b) Describe how to implement CountCrossInversions($A[1 .. n], m$) to run in $O(n)$ time ***assuming the subarrays $A[1 .. m]$ and $A[m+1 .. n]$ are sorted***. You may now access members of A directly. (This problem would be very difficult if you could not!) *[Hint: Modify the Merge procedure from mergesort.]*

   (c) Now, use the answers for the previous parts to describe an algorithm for counting inversions in an n-element array. It is fine, but not necessary, for your algorithm to change A in some way. Again, justify correctness, but don't worry about running time yet. *[Hint: Modify mergesort.]*

   (d) Hopefully, your algorithm runs in $O(n \log n)$ time. Explain the running time of your algorithm. *[Hint: You could use any one of recursion trees, the master method, or an appeal to how similar it is to an algorithm you saw in class.]*

3. Tomorrow is the big dancing contest you've been training for your entire life. You've obtained an advance copy of the list of n songs that the judges will play during the contest, in chronological order.

   You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know if that you dance to the kth song on the schedule, you will be awarded exactly $Score[k]$ points, but then you will be physically unable to dance for the next $Wait[k]$ songs (that is, you cannot dance to songs $k+1$ through $k+Wait[k]$). The dancer with the highest total score at the end of the night wins the contest, so you want your total score to be as high as possible.

   Our goal is to design an efficient dynamic programming algorithm to compute the maximum total score you can achieve. The input to this sweet algorithm is the pair of arrays $Score[1 .. n]$ and $Wait[1 .. n]$.

(a) Finding the right recursive structure for the problem is the hardest part. As a first attempt, let's try the following idea: We need to find a sequence of songs to dance to, so we should commit to dancing some song i and then guess the next song we should (and can) dance to. ***Specify*** a function, based on the above idea, that we would want to solve recursively. ***Do not describe a recurrence for this function yet.*** *[Hint: Look at the definition of $LISfirst(i)$ given in Erickson 3.6, page 13.]*

(b) Derive a recurrence for your function. Don't forget the base case(s).

On second thought, that strategy might be too slow.[1] Instead, let's consider the input sequence of songs and decide *if* you should dance to the first remaining song based on our past decisions.

For any i where $1 \le i \le n+1$ let $MaxTotal(i)$ be the maximum total score you can achieve dancing to songs i through n (if $i = n+1$, then there are no songs left to dance to). You can either dance song i or skip it, so

$$MaxTotal(i) = \begin{cases} 0 & \text{if } i = n+1 \\ \max\left\{ \begin{array}{r} Score[i] + MaxTotal(i + Wait[i] + 1), \\ MaxTotal(i+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

(c) In what kind of ***memoization data structure*** should we store the solutions to all subproblems $MaxProblem(i)$?

(d) What is a good ***evaluation order*** for solving the subproblems so each subproblem is solved after the ones it is dependent upon?

(e) What will be the final ***space*** and ***time*** complexity of the dynamic programming algorithm? *[Hint: This part may be easier after solving part (f), but you should be able to do it now using only the recurrence.]*

(f) Write the iterative algorithm that computes the maximum possible score you can achieve. Don't forget to return the maximum possible score after filling your data structure.

[1] The solutions we have in mind would lead to an $O(n^2)$ time algorithm.