Exam 2

CS 3510A, Spring 2014

These solutions are being provided **for your personal use only**. They are not to be shared with, or used by, anyone outside this class (Spring 2014 section of Georgia Tech CS 3510A). Deviating from this policy will be considered a violation of the GT Honor Code. **Instructions:**

- **Do not open this exam packet until you are instructed to.**
- **Write your name** on the line below.
- You will have **50 minutes** (from 11:05 to 11:55) to earn **50 points**. Pace yourself!
- You may use one double-sided 8.5-by-11" study sheet. **You may not use any other resources, including electronic devices.**
- Do your best to fit your final answers into the space provided below each question. You may use the backs of the exam pages, and the final page of the exam, as scrap paper.
- **You may refer to any facts from lectures or the textbook without re-deriving them.**
- Your work will be graded on correctness and clarity. Please write legibly!

Question	Points	Score
1	10	
2	20	
3	20	
Total:	50	

Your name: _____

1. (10 points) *True or False.* State whether each of the following sentences is true or false, **and give a convincing justification for your answer**. A correct justification is worth more than a correct true/false answer.

   (a) If Quicksort uses a deterministic linear-time algorithm to find the median of its input (sub)array and partitions around the median, then Quicksort has $O(n \log n)$ worst-case runtime.

   Solution: True. Partitioning around the median in this way yields a worst-case runtime of $T(n) = 2T(n/2) + O(n)$, which solves to $T(n) = O(n \log n)$.

   (b) A "top-down" recursive algorithm can never take more time than the corresponding "bottom-up" dynamic programming algorithm.

   Solution: False. A recursive algorithm (without memoization) will often take *much more* time than a bottom-up one—sometimes *exponentially* more—because it can end up recomputing the answers to the same subproblems multiple times.

2. (20 points) *Order statistics.*

(a) Describe an $O(n)$-time algorithm that, given an array A of $2n+1$ distinct elements and an integer $k \leq n$, outputs *in some order* the $2k+1$ "middle" elements of A, i.e., the $(n+1-k)$th through $(n+1+k)$th order statistics. Briefly justify the correctness and runtime of your algorithm. (Remember that you can rely on algorithms we've covered in class.)

> **Solution:** There are at least two kinds of correct answers.
>
> Solution 1: First, use Select to choose the $(n+1+k)$th order statistic (call it y) and again to choose the $(n+1-k)$th order statistic (call is x), each in time $O(n)$. Then make a pass through the array and output any element $A[i]$ that is between x and y (inclusive), i.e., $x \leq A[i] \leq y$. This phase also takes time $O(n)$, and works because all the array elements are distinct.
>
> Solution 2, which additionally places all the desired elements in the "middle" of the array: First, use Select to choose the $(n+1+k)$th order statistic (call it y) and partition the array using y as a pivot. Next, use Select to choose the $(n+1-k)$th order statistic (call it x) of the left subarray (i.e., $A[1 \ldots (n+k)]$, which contains all the elements $< y$), and partition *just the left subarray* around x. Finally, output all the elements from $A[(n+1-k) \ldots (n+1+k)]$, which contains all the elements that are $\geq x$ and $\leq y$, as desired. The total runtime is $O(n)$ because we only make two calls each to Select and Partition, plus the final pass through the middle of the array.
>
> A common incorrect answer was to find the median m of A, then build an array of differences $D[i] = A[i] - m$ (possibly taking absolute values), and use the differences somehow to output the desired elements. In general this does not work (or at best it is useless work), because the *values* of the differences have no effect on the elements we want to find. All we want are the k largest values smaller than the median, and the k smallest values larger than the median; it does not matter *how far* those values are from the median.

(b) Describe an algorithm that, given an array of n distinct elements, outputs the $(n/2)$th, $(n/4)$th, $(n/8)$th, $(n/16)$th, etc. order statistics (so $\log_2 n$ elements in all), in a total of $O(n)$ time. Briefly justify the correctness and runtime of your algorithm.

> **Solution:** First find the median (the $(n/2)$th order statistic) using a linear-time algorithm, output the median, and partition around it. Then recurse on *just the left half subarray* (of length $n/2$) to output the remaining desired elements. This algorithm is correct because the elements in the left half subarray are the smallest $n/2$ elements of the original array, and by induction the recursive call outputs the $(n/2)/2 = (n/4)$th, $(n/2)/4 = (n/8)$th etc. order statistics of the left half subarray, which are the same order statistics of the full array. The runtime recurrence is $T(n) = T(n/2) + \Theta(n)$, which solves to $T(n) = \Theta(n)$ by the master theorem.

3. (20 points) *A jump game.* Consider the following game: given an array $A[1 \ldots n]$ of non-negative integers, you start at $A[1]$, the first position of the array. Each element $A[i]$ in the array represents the *maximum* number of positions you can "jump" forward in the array from position i. The goal is to reach the final position $A[n]$ of the array, if possible, by taking legal jumps. Since it is meaningless to jump past the end of the array, we require that none of the array entries are large enough to allow this.

   For example, for $A = [2, 3, 1, 1, 0]$ one can win in at least two ways: by using the "greedy" strategy of always jumping by the maximum allowed number of positions, ***OR*** by jumping from $A[1] = 2$ to $A[2] = 3$ to $A[5]$. However, for array $A = [3, 2, 1, 0, 0]$ there is no way to win.

   (a) Give an example of an array A where the "greedy" strategy does *not* win the game, but some other strategy does, and show a winning strategy. (Remember that the entries of A cannot allow for jumping past the end of the array, so, for example, $A = [4, 0]$ would not be a correct answer.)

   Solution: For array $A = [2, 2, 0, 0]$ the "greedy" strategy will choose to jump from $A[1] = 2$ to $A[3] = 0$, and then will be forced to stop without winning. However, a winning strategy is to jump from $A[1] = 2$ to $A[2] = 2$ to $A[4]$.

   (b) Let $W[i]$ denote the boolean value indicating whether it is possible to win the game when starting at $A[i]$. Write the base case and a recursive expression for the entries of W.

   Solution: Clearly, we can always win if we start at $A[n]$ (this is the base case). For $i < n$, we can win from $A[i]$ if and only if we can with from any of $A[i+1], A[i+2], \ldots, A[i + A[i]]$ (where this list is empty if $A[i] = 0$), which are the only locations we are allowed to jump directly to from $A[i]$. In summary:

   $$W[n] = \text{true}$$
   $$W[i] = W[i+1] \vee W[i+2] \vee \cdots \vee W[i + A[i]].$$

   (c) State, with brief justification, the worst-case runtime of the dynamic programming algorithm based on your expressions above. Give your answer in $\Theta(\cdot)$ notation in terms of n, the length of A.

   Solution: The algorithm fills in W from index n down to 1. When setting $W[i]$ it must inspect up to $n - i$ other positions of W, doing constant work for each. So its total runtime is $1 + 2 + \cdots + (n-1) = \sum_{i=1}^{n-1} i = \Theta(n^2)$.

   (d) Describe what extra information to compute and store, in addition to $W[1 \ldots n]$, that allows reconstructing a winning strategy (when one exists) in just $\Theta(n)$ additional time. Computing the extra information should not increase the asymptotic runtime of the whole algorithm.

   Solution: We use an array $L[1 \ldots n]$, and when computing $W[i]$ we set $L[i] \in \{i+1, i+2, \ldots, i + A[i]\}$ to be any of the legal targets for which $W[L[i]] = \text{true}$ (if one exists). Such a location is found as a side effect whenever we assign $W[i] = \text{true}$.

   If the game is winnable (i.e., if $W[1] = \text{true}$), then we just follow the indices $L[1]$, $L[L[1]]$, etc. to reconstruct a winning strategy in $O(n)$ time.

Scrap paper — no exam questions here.