Exam 1

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	20	
2	15	
3	15	
Total:	50	

Your name: _____

1. (20 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) The Master Theorem can be used to solve the recurrence $T(n) = 16T(n/4) + \Theta(n)$, and yields the solution $T(n) = \Theta(n^2)$.

   > **Solution:** True. Set $b = 4$ and $a = 16$, so $n^{\log_b a} = n^2$. Since $f(n) = \Theta(n) = O(n^{2-\epsilon})$ for $\epsilon = 1 > 0$, case 1 of the Master Theorem applies and yields the stated solution.

   (b) If $f(n) = \Theta(n^2)$, then $f(n) = \omega(n)$.

   > **Solution:** True. From $f(n) = \Theta(n^2)$, we know $f(n) = \Omega(n^2)$, which means there exists some constant $c > 0$ such that $f(n) \geq c \cdot n^2$ for all large enough n. Therefore, $f(n)/n \geq c \cdot n$ for all large enough n, and so $\lim_{n\to\infty} f(n)/n \geq \lim_{n\to\infty} cn = \infty$. This means $f(n) = \omega(n)$.

   (c) There is an input array of n distinct elements that forces RandomizedQuicksort to run in $\Omega(n^2)$ time.

   > **Solution:** False. We proved in class that on *any* input array (of distinct elements), the *expected* runtime of RandomizedQuicksort is only $O(n \log n)$. While some very unlucky choices of random pivots can cause it to run in $\Omega(n^2)$ time, these random choices are made *by the algorithm*, and cannot be forced upon it by any particular input array.

   (d) It is possible to alphabetically sort a list of n English words, each of length at most 20 letters, in $O(n)$ time.

   > **Solution:** True. Use radix sort on the individual letters (from last to first), after appending spaces to make all the words have length exactly 20. The number of letters in the alphabet is $k = 26 = O(1)$ (or $k = 52 = O(1)$ counting capital letters), and the words have length $d = 20$, so the total runtime of radix sort is $O(d(n + k)) = O(20(n + 26)) = O(n)$.

2. (15 points) *Fixed points.* In an array $A[1 \ldots n]$ of integers, a *fixed point* is an index $i \in \{1, \ldots, n\}$ such that $A[i] = i$.

   (a) Let $A[1 \ldots n]$ be a *sorted* array of *distinct* integers. Supposing that $A[j] > j$ for some index j, state with brief justification what portion of the array *cannot* contain a fixed point. What if $A[j] < j$?

   Solution: If $A[j] > j$, then $A[k] > k$ for every $k > j$ as well. This is because the array elements are distinct, so $A[i+1] \geq A[i] + 1$ for all i, and hence $A[k] \geq A[j] + (k - j) > j + (k - j) = k$. Therefore, if $A[j] > j$, then there cannot be any fixed point in $A[j \ldots n]$. By similar reasoning, if $A[j] < j$, then there cannot be any fixed point in $A[1 \ldots j]$.

   (b) Using your observations from above, write pseudocode for a recursive divide-and-conquer algorithm FixedPoint$(A[1 \ldots n])$ that, given a sorted array of distinct integers, outputs a fixed point i if one exists; otherwise, it should output "none." (Make your algorithm as fast as you can.)

   Solution: We write a generic procedure FixedPoint for a subarray $A[\ell \ldots r]$, and call it on $A[1 \ldots n]$, i.e., with $\ell = 1$ and $r = n$.

```
FixedPoint(A[ℓ ... r])
    if (ℓ > r)
        return "none"
    m = ⌊(ℓ+r)/2⌋
    if (A[m] == m)
        return m
    else if (A[m] < m)
        return FixedPoint(A[(m + 1) ... r])
    else
        return FixedPoint(A[ℓ ... (m − 1)])
```

   (c) Write a recurrence for your algorithm's worst-case runtime $T(n)$, and give a closed-form solution in $\Theta(\cdot)$ notation (using any valid method).

   Solution: Since the algorithm does constant work (computing the middle index m and comparing m with $A[m]$) plus one recursive call on a subarray of roughly half the size, we have that

$$T(n) = T(n/2) + \Theta(1).$$

   Using the Master theorem or a recurrence tree, this recurrence solves as $T(n) = \Theta(\log n)$.

3. (15 points) *Merging multiple lists.*

(a) Briefly but clearly describe *in words* (no pseudocode needed) an algorithm 3Merge(A, B, C) that merges three sorted arrays A, B, C of total length n into one sorted list, and state its runtime in $\Theta(\cdot)$ notation.

Solution: The 3Merge algorithm would work similarly to the algorithm Merge used as a subroutine in MergeSort. We maintain three index pointers i, j and k into A, B and C respectively, and start with an empty array D and index ℓ. In each iteration, we compare the elements $A[i]$, $B[j]$ and $C[k]$, copy their minimum to $D[\ell]$ and increment ℓ, also incrementing the index of the array we copied from. When one index reaches the end if its array, we continue by comparing the other two, and when the second one reaches the end, we copy the rest of the elements from the third array into D, which we return as the solution. The runtime of this algorithm is $\Theta(n)$ because we do a constant number of comparisons and other operations in order to place each element in D.

Alternatively, we can run Merge(Merge$(A, B), C)$ to get the same asymptotic runtime.

(b) Consider a variant of MergeSort that divides its input into three nearly equal-size subarrays, sorts them recursively, and then merges them using your algorithm 3Merge. Write a recurrence for the worst-case runtime of this algorithm on an array of n elements, and give a closed-form solution in $\Theta(\cdot)$ notation (using any valid method).

Solution: The recurrence is $T(n) = 3T(n/3) + \Theta(n)$. By the Master Theorem, this solves as $T(n) = \Theta(n \log n)$.

(c) Now consider the problem of merging k sorted arrays of total length n into one sorted list.

i. One algorithm for this problem is the natural generalization of Merge and 3Merge. How much time, in $\Theta(\cdot)$ notation as a function of k and n, does this algorithm take? (You do not need to justify your answer.)

Solution: The runtime of this algorithm is $\Theta(kn)$, because when placing each element in the output array it does $\Theta(k)$ comparisons to find the minimum element among all k that are being considered in the k lists.

ii. Write pseudocode for an algorithm that takes only $\Theta(n \log k)$ time, and justify why it runs in this time.

Solution: The algorithm uses the usual Merge routine as a subroutine.

MultiMerge$(A_1, \ldots, A_k)$
  if $(k == 1)$ return A_1
  $B =$ MultiMerge$(A_1, \ldots, A_{k/2})$
  $C =$ MultiMerge$(A_{k/2+1}, \ldots, A_k)$
  return Merge(B, C)

(We did not ask you to do the following correctness and runtime analysis, but present it below for your interest.)

Correctness follows easily by induction: the base case is obviously correct, and in the recursive case, B and C are sorted arrays of the elements in the first and last $k/2$ arrays (respectively), and Merge merges them into a single sorted array.
The runtime $T(n, k)$ is a function of two variables, n and k. The runtime recurrence is

$$T(n, k) = T(n_1, k/2) + T(n_2, k/2) + \Theta(n),$$

where n_1, n_2 respectively denote the total number of elements in first and last $k/2$ arrays, and $n = n_1 + n_2$. Using a recurrence tree, the amount of work at each level is $c \cdot n$ for some constant c, and there are $\lg(k)$ levels, so the solution is $T(n, k) = \Theta(n \log k)$.

Scrap paper — no exam questions here.