CSE 202: Midterm
May 3, 2005—Day 11

Name: ____________________________
Student ID:_______________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/5</td>
</tr>
<tr>
<td>2</td>
<td>/5</td>
</tr>
<tr>
<td>3</td>
<td>/10</td>
</tr>
<tr>
<td>4</td>
<td>/15</td>
</tr>
<tr>
<td>5</td>
<td>/15</td>
</tr>
<tr>
<td>6</td>
<td>/15</td>
</tr>
<tr>
<td>7</td>
<td>/15</td>
</tr>
<tr>
<td>8</td>
<td>/20</td>
</tr>
<tr>
<td>Total</td>
<td>/100</td>
</tr>
</tbody>
</table>
1. 5 pts. Which of the following statements follows from a decision-tree analysis of sorting complexity?

(a) No algorithm of any kind can sort any kind of input in less than $\Omega(n \log n)$ time.

(b) Algorithms that do only comparisons between elements require at least $\Omega(n \log n)$ comparisons for all inputs

(c) All paths from the root to a leaf in a comparison-based decision tree have length at least $n \log n$

(d) No paths from the root to a leaf in a comparison-based decision tree have length greater than $n \log n$

(e) All of the above

(f) None of the above

f) None of the above. a) isn’t true because we know that counting sort can sort integers from 1..n in linear time. b) isn’t true because some inputs can be sorted in less than $n \log n$ time. c) is false for the same reason as b). d) is false because our analysis didn’t come up with an upper-bound on the number of comparisons.

2. 5 pts. We want to prove that any comparison-based sorting algorithm requires at least $n - 1$ comparisons for all inputs of the form $A[1..n]$. Which of the following is the idea of a correct proof:

(a) If we look at the sorted list $A'[1..n]$, we must have compared all adjacent elements $A'[j], A'[j + 1]$. Otherwise, the relative ordering between the two wasn’t determined.

(b) For all inputs, if we compare $A[1]$ to $A[2]$, and, in general, $A[i - 1]$ to $A[i]$, we need at least $n - 1$ comparisons. Otherwise one of the elements won’t be compared, and could be less than all others, or greater than all others.

(c) The best case input is if it is already sorted. All that is needed is to verify elements are monotonically non-decreasing. That requires $n - 1$ comparisons.

(d) Insertion sort requires at least $n - 1$ comparisons for all inputs.

(e) None of the above
a) By looking at the sorted list, we can argue backward about what comparisons any algorithm must have made to correctly sort the list. All the other answers assume how a particular algorithm is working.

3. 10 pts. We want to generate a uniform random permutation of an array, $A[1..n]$. Is it sufficient to show that for each element $A[i]$, the probability that our algorithm places it in position $j$ is $1/n$? Why or why not?

No. Assuming our algorithm generates permutations, it is not sufficient. For example, given $A = [1, 2, 3]$, our algorithm could generate, with equal probability, $[1, 2, 3]$, $[2, 3, 1]$, and $[3, 1, 2]$. Each element appears at a particular position with equal probability, but no all permutations are being generated.

4. 15 pts. What is a randomized algorithm? Reconcile the fact that the expected running time of Randomized-Quicksort is $\Theta(n \log n)$ with the existence of an adversary that can force Randomized-Quicksort to take $\Theta(n^2)$ time.

A randomized algorithm is an algorithm that uses randomness, in the form of making calls to a random-number generator.

The expected running time of Randomized-Quicksort is based on the assumption that the input is independent of the random choices that the algorithm makes. However, the adversary chooses the input dynamically, after random choices are being made, and so the input is not independent of the random choices.

5. 15 pts. Define runtime, worst-case runtime, expected runtime, and average runtime.

Runtime is the number of operations an algorithm takes on a particular input.

Worst-case runtime is the maximum number of operations an algorithm takes over a set of inputs (usually all inputs of a given size).

Expected runtime measures the runtime of a randomized algorithm by calculating the expectation of the runtime over the random choices made by the algorithm.
Average runtime is number of operations an algorithm takes over a set of inputs (usually all inputs of a given size) divided by the number of inputs.

6. 15 pts. What does an amortized runtime analysis of an algorithm show?

An amortized runtime analysis shows the worst-case runtime for a sequence of operations rather than a single operation.

7. 15 pts. Draw a sorting network that will sort 4 numbers. Prove that it is correct.

(The answer should have been drawn, but in this solution is just described). With four inputs, a, b, c, d on lines \( l_1, l_2, l_3, l_4 \), compare \((l_1, l_2), (l_2, l_3), (l_3, l_4)\). At this point, the maximum element is on line \( l_4 \). Now, compare \((l_1, l_2), (l_2, l_3)\). At this point, the second-highest element is on line \( l_3 \). Now, compare \((l_1, l_2)\). At this point, the lowest and second-lowest elements are on lines \( l_1 \) and \( l_2 \).

8. 20 pts. You have a collection of \( n \) uncolored jellybeans of various flavors and you’d like to know whether there’s a subset of those jellybeans that are the same flavor. However, you can’t directly check the flavor of a jellybean. Instead, you can provide two jellybeans to your hamster, who will lick each jelly bean, and then run in circles if the jellybeans have the same flavor, or roll over if the jellybeans have a different flavor. Provide an algorithm that determines whether more than \( n/2 \) of the jellybeans are the same flavor. The total number of jellybean pair tests done by your hamster should be in \( O(n \log n) \).

Solution: use divide and conquer to recursively solve a more general problem: not just knowing whether there is such a subset, but finding a representative element of the subset.

Two base cases are easy: given a set of size 0, there is no such subset (and no element). Given a set of size 1, there is such a subset (and the single jellybean is the representative).

Given a set of size \( n \), break it into two subset of size \( n/2 \), run the algorithm recursively. If no representative is found from either subset, then there is not a large-enough subset (because if the subset is \( > n/2 \), then \( > n/4 \) of the elements from one of the subsets must have
the same flavor). Check the representatives against all elements of the set to see whether one or the other is a representative of a large-enough subset. If so, return it; if not, return NULL

FindRepresentativeOfBigSubset(L)
begin
if L has zero elements, return NULL
if L has one element, return it.
L1 = first half of L
L2 = second half of L
T1 = FindRepresentativeOfBigSubset(L1)
T2 = FindRepresentativeOfBigSubset(L2)
S1 = S2 = empty set
if T1 is not NULL
   S1 = set of elements which have the same flavor as T1
if T2 is not NULL
   S2 = set of elements which have the same flavor as T2
if |S1| > |L|/2 then
   return T1
if |S2| > |L|/2 then
   return T2
return NULL
end

Runtime analysis: the recurrence is of the form: \( T(n) = 2T(n/2) + O(n) \) (since at most 2\( n \) jellybean tests are done by the algorithm at any level). By the master method, \( T(n) = O(n \log n) \)