Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains 6 multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains 16 pages, including this one and two extra sheets of scratch paper, which are included for your convenience.
- This quiz is closed book. You may use one double sided Letter (8⅓” × 11”) or A4 crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time re-deriving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Generally, a problem’s point value is an indication of how much time to spend on it.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
<th>Grade</th>
<th>Initials</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>80</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name: ________________________________

Circle your recitation time:

F10   F11   F12   F1   F2   F3
Problem 1. [1 points] Write your name on every page! Don’t forget the cover.

Problem 2. Recurrences [15 points] (5 parts)

Solve the following recurrences by giving tight $\Theta$-notation bounds. You do not need to justify your answers, but any justification that you provide will help when assigning partial credit. As usual, assume that for $n \leq 10$, $T(n) = O(1)$.

(a) [2 points] $T(n) = 3T(n/5) + \lg^2 n$

Solution: Since $\lg^2 n = O(n^{\log_2 5 - \epsilon})$, this is the Case 1 of the Master Method. We have $T(n) = \Theta(n^{\log_5 3})$.

(b) [2 points] $T(n) = 2T(n/3) + n \lg n$

Solution: Since $n \lg n = \Omega(n^{\log_2 3 + \epsilon})$ and $2^{\frac{2}{3}} \lg^{\frac{2}{3}} \leq \frac{2}{3} n \lg n$, this is the Case 3 of the Master Method. We have $T(n) = \Theta(n \lg n)$. 
(c) [2 points] \( T(n) = T(n/5) + \log^2 n \)

**Solution:** Since \( \log^2 = \Theta(n^{\log_5 1}) \log^2 n \), this is the Case 2 of the Master Method. We have \( T(n) = \Theta(\log^3 n) \).

(d) [3 points] \( T(n) = T(n - 2) + \log n \)

**Solution:** \( T(n) = \Theta(n \log n) \). This is \( \sum_{i=1}^{n/2} \log 2i \geq \sum_{i=1}^{n/2} \log i \geq (n/4)(\log n/4) = \Omega(n \log n) \). For the upper bound, note that \( T(n) \leq S(n) \), where \( S(n) = S(n-1) + \log n \), which is clearly \( O(n \log n) \).
(e) [6 points] An improvement to the multiplication method given in class involves splitting each \( n \)-bit number into three pieces of \( n/3 \) bits each (i.e., write \( X \) as \( 2^{2n/3}A + 2^{n/3}B + C \) and write \( Y \) as \( 2^{2n/3}D + 2^{n/3}E + F \)). A straightforward product would now involve 9 multiplications of \( n/3 \)-bit numbers, but by cleverly rearranging terms, it is possible to reduce this to some number \( a \) multiplications of \( n/3 \)-bit numbers, plus a constant number of additions and shifts.

- Set up and solve the recurrence for the improved running time algorithm in terms of \( n \) and \( a \).
- What is the largest value of \( a \) such that this algorithm is asymptotically faster than the algorithms we learned in class, which ran in \( O(n^{1.58}) \)?

**Solution:** In order to beat the algorithm in class,

\[
\log_3 a < \log_2 3 \\
\lg a \lg 2 < \lg^2 3 \\
a \leq 5
\]
Problem 3. True or False, and Justify [27 points] (9 parts)

Circle T or F for each of the following statements, and briefly explain why. Your justification is worth more points than your true-or-false designation.

(a) T F [3 points] The sum of the largest $\lg n$ elements in an unsorted array of $n$ distinct elements can be found in $O(n)$ time.

Solution: True. We can use deterministic select algorithm to find the largest $\lg n$ elements in $O(n)$. Compute sum of these number can be done in $O(\lg n)$ time.

(b) T F [3 points] A Monte Carlo algorithm for every possible input and every sequence of random coin tosses always returns a correct output. However with some nonzero probability over the coin tosses a Monte Carlo algorithm may not run in polynomial time.

Solution: False. Monte Carlo algorithms may return incorrect outputs with some probability.
(c) T F [3 points] Let \( N \) be a positive integer and let \( a \) be an element in \( \mathbb{Z}_N^* \). If \( a \) has at least three different square roots modulo \( N \), then \( N \) is composite.

**Solution:** True. We can prove this claim by contradiction. Assume that \( N \) is a prime. Consider any integer \( a \in \mathbb{Z}_N^* \), such that \( a \) has \( x \) as a square root modulo \( N \).

For any \( y \neq x \) and \( y \neq -x \), \( y^2 \neq x^2 = a \) modulo \( N \) (shown in Homework 2-2).

Therefore, \( a \) has at most two different square roots modulo \( N \). This contradicts the fact that \( a \) has at least three different square roots modulo \( N \). Therefore, \( N \) is composite.

(d) T F [3 points] An adversary can construct an input of length \( n \) to force RANDOMIZED-QUICKSORT to run in \( \Omega(n^2) \) time.

**Solution:** False. The expected running time of RANDOMIZED QUICKSORT is \( \Theta(n \log n) \). This applies to any input.
(e) T F [3 points] Searching in a skip list takes $\Theta(\log n)$ time with high probability, but could take $\Omega(2^n)$ time with nonzero probability.

Solution: True. A skip list could be of any height or be a simple linked list, depending on its random choices.

(f) T F [3 points] If an operation runs in $O(f(n))$ amortized time, it also takes $O(f(n))$ worst-case time (per operation).

Solution: False. Consider the table doubling example in the lecture. Although the amortized cost of each operation in this example is $O(1)$ but the worst-case time is $O(n)$. 
(g) T F  [3 points] Let k, i, j be integers, where k > 3 and 1 ≤ i, j ≤ k. Let $h^k_{ij}$ be the hash function mapping a k-bit integer $b_1b_2\ldots b_k$ to the 2-bit value $b_ib_j$. For example, $h^8_{31}(00101011) = 10$.

The set $\{h^k_{ij} : 1 ≤ i, j ≤ k\}$ is a universal family of hash functions from k-bit integers into $\{00, 01, 10, 11\}$.

Solution: False. Take $x = 0, y = 1$. Then all of $x$’s binary digits are the same as $y$’s, except for the least significant bit. Thus, if we choose one of the $k$ hash functions at random, $\Pr[h(x) = h(y)] = (k^2 - k)/k^2 = (k - 1)/k > 1/4$. If this were a universal class of hash functions, then this probability should be at most $1/4$.

(h) T F  [3 points] The rightmost child subtree of the root of an $n$-node 2-3 tree contains $\Omega(n)$ nodes.

Solution: False. A tree with all degree-3 nodes on one subtree and degree-2 nodes on the other will have depth $h = \log_3 n$. There will be $2^{\log_3 n} = n^{\log_3 2} = o(n)$ nodes on the sparse subtree.
(i) T F [3 points] The following Monte Carlo primality testing algorithm has error probability less than $\frac{1}{2}$.

Algorithm FERMAT-TEST. On input $N > 2$:

1. Repeat the following twice:
   (a) Pick a random integer $A \in \mathbb{Z}_N^*$.
   (b) If $A^{N-1} \not\equiv 1 \pmod{N}$, then return “composite”.

2. If the algorithm did not return “composite” in step 1, return “probably prime”.

Solution: False. If $N$ is a Carmichael number, the algorithm always returns “probably prime”. Therefore, the error probability for the case that $N$ is a Carmichael number is 1.
Problem 4. Short Answer [19 points] (4 parts)

Give brief, but complete, answers to the following questions.

(a) [5 points] You are maintaining a collection of linked lists that support the following operations:

- \(\text{INSERT}(\text{item, list}): \) insert item into list (cost = 1).
- \(\text{SUM}(\text{list}): \) sum the items in list, and replace the list with one item that is the sum (cost = length of list).

Show that, starting with empty lists, we can assign an amortized cost of 2 to each \(\text{INSERT}\) and an amortized cost of 1 to each \(\text{SUM}\).

**Solution:** We’ll maintain the invariant that every item has one credit. Insert gets 2 credits, which covers one for the actual cost and one to satisfy the invariant. Sum gets one credit, because the actual cost of summing is covered by the credits in the list, but then the result of the sum will need one credit to maintain the invariant.

Many people confused credit arguments and potential arguments and tried to use a potential function even though one was never defined. These are equivalent, but different. You should only use one at a time.

(b) [4 points] Suppose we insert an element into a 2-3 tree, and the \(\text{INSERT}\) algorithm splits \(k\) nodes. Give an exact (not big-O) upper bound on the number of nodes in the tree that are created or modified in this case.

**Solution:** Each split of node \(x\) creates one new node (\(x\)’s sibling) and modifies one (i.e. \(x\)), so two nodes are created / modified for each split. The split also results promoting a key to \(x\)’s parent, which may further cause the parent node to split. If the parent node split, the modification to the parent is already accounted for in the splitting of the parent node. Thus, two nodes are created / modified for each split. On the other hand, we need to account for an additional modification to the final node (closest to the root) where the split stops. Or, if the split continues all the way to the root, a new root is created. Therefore, for \(k\) splits, a total of \(2k + 1\) nodes are created / modified. Alternatively, when a node \(x\) split, one may delete \(x\) entirely, and replace \(x\) with two new nodes (each getting half of \(x\)’s keys). In which case, three nodes are created / modified / deleted, and the answer is then \(3k + 1\).
(c) [6 points] Suppose that you are given an array $A$ of $n$ bits that either contains all zeros or contains $2n/3$ zeros and $n/3$ ones in some arbitrary order. Your goal is to determine whether $A$ contains any ones.

1. Give an exact lower bound in terms of $n$ (not using asymptotic notation) on the worst-case running time of any deterministic algorithm that solves this problem.
2. Give a randomized algorithm that runs in $O(1)$ time and gives the right answer with probability at least $1/3$.
3. Give a randomized algorithm that runs in $O(1)$ time and gives the right answer with probability at least $5/9$.

Solution:

1. Any correct deterministic algorithm must look at $2n/3 + 1$ entries, because if it didn’t, it could see all zeros even when there was a one somewhere.

2. Pick a random location. If it is a 1, output “has ones”. Otherwise, output “doesn’t have ones”. Clearly the algorithm runs in $O(1)$ time. If the array is all zeros, it is always correct. However if the array contains $1/3$ ones, the algorithm may make a mistake. Since there are $n/3$ ones, you will select a 1 value at random and correctly output “has ones” with probability $1/3$.

3. Pick two random locations. The correctness and $O(1)$ runtime are identical as above. Again, if the array is all zeros, the algorithm is always correct. A mistake arises if the array contains ones, but the algorithm picks two zeros. Picking a single zero occurs with probability $2/3$, and picking two zeros independently occurs with probability $(2/3)^2 = 4/9$. Therefore, the probability that the algorithm is correct is $1 - (4/9) = 5/9$. 


(d) [4 points] Show that, if all edge weights in a graph are distinct, then the minimum spanning tree is unique.

**Solution:** Suppose (for contradiction) that there were two different MSTs $T_1$ and $T_2$ of a graph $G = (V, E)$ with distinct edge weights; let $e$ be the lightest edge in $E(T_1) \setminus E(T_2) \cup E(T_2) \setminus E(T_1)$, where $E(T)$ denotes the set of edges in a tree $T$. Wlog, assume that $e \in T_1$. Consider the graph $G' = (V, E(T_2) \cup \{e\})$. $G'$ has a cycle that contains $e$. In this cycle, there must be an edge $e' \in E(T_2) \setminus E(T_1)$ (otherwise $T_1$ will contain the cycle). By the definition, $e'$ must have greater weight than $e$. Consider the graph $G'' = (V, E(T_2) \cup \{e\} \setminus \{e'\})$. $G''$ is connected and has exactly $n - 1$ edges. Therefore, $G''$ is a MST. Also, the total weight of $G''$ is less than $T_2$, a contradiction.
Problem 5. Tree cover [9 points] Let $T$ be a rooted tree with $n$ nodes $\{1, 2, \ldots, n\}$ and parent array $P[1, 2, \ldots, n]$ where $P[i] = j$ if the node $j$ is the parent of $i$, and $P[i] = i$ if $i$ is the root of the tree. Thus the edge set of the tree is $\{\{i, P[i]\} \mid P[i] \neq i\}$.

We say that a vertex $c$ covers an edge $e = \{u, v\}$ if $c$ is one of the endpoints of $e$, that is, $c = u$ or $c = v$. The vertex cover problem is the following: given a tree $T$, find a minimum-size subset of vertices $S \subseteq \{1, 2, \ldots, n\}$ such that $S$ covers all edges, i.e., every edge $\{i, P[i]\}$ in the tree is covered by at least one vertex in $S$.

Give a polynomial-time greedy algorithm to solve the problem above and prove its correctness. You may assume that each node $x$ has an attribute $x$.depth that stores the depth of $x$ in the tree.

Solution: Our algorithm works as follows:

1. Sort all the node in a nonincreasing order of depth.
2. Set $S$ to $\emptyset$.
3. Scan through all the nodes in the sorted array, for a node $x$, if the edge $(x, P[x])$ has not been covered, add $P[x]$ (x’s parent node) into $S$.

Running time analysis: This algorithm takes $O(n \lg n)$ to sort, and $O(n)$ to scan through the sorted array (i.e., each step in the scan can be implemented in $O(1)$ time). Therefore, the total running time is $O(n \lg n)$. The running time can be improved to $O(n)$ by using COUNTING-SORT (see book).

Proof of correctness: Observe that at the time when we consider a vertex $x$ in our algorithm, all edges incident to vertices before $x$ are already covered. When we consider a vertex $x$, if edge $(x, P[x])$ which has not been covered, in order to cover this edge we have to either pick $x$ or $P[x]$ (picking both is wasteful).

1. Greedy choice property: Picking $P[x]$ is the best option since $P[x]$ covers as least as many (uncovered) edges as $x$ does.
2. Optimal substructure: $P[x]$ covers all (uncovered) edges that $x$ covers (which is $(x, P[x])$). Therefore, if there is an optimal solution that pick $x$, there should be another optimal solution that replaces $x$ by $P[x]$. Therefore, picking $P[x]$ should lead to an optimal solution.
Problem 6. Average in 2-3 Trees [9 points] (2 parts) You are the IT consultant who is managing the salary database for a big firm. The firm salary database is designed as a 2-3 tree so that it is efficient to Search, Insert, or Delete by salary in which each node in the tree stores the salary for one or two employee (as in regular 2-3 tree).

Every month the firm has to submit a report to the Department of Fairness about the salary for low-income employees in the firm. In particular, the report should contain the average salary of all employees with salary at most \( x \) for some \( x > 0 \). In order to compile the report efficiently, you are asked to implement a new operation \( \text{AVERAGE}(x) \) in the database which returns the average salary of all employees whose salary is at most \( x \).

(a) [6 points] What extra information needs to be stored at each node? Describe how to answer an \( \text{AVERAGE}(x) \) query in \( O(\log n) \) time using this extra information.

**Solution:** Each node \( x \) should store \( x\.size \) - the size of the subtree rooted at \( x \) - and \( x\.sum \) - the sum of all the key values in the subtree rooted at \( x \). For a value \( x > 0 \), let \( S_x \) be the set of all keys less than or equal to \( x \). Let \( A_x \) and \( B_x \) be the sum and the size of \( S_x \).

We can compute \( A_x \) as follows. Let \( u \) be the leaf with smallest key larger than \( x \). Finding \( u \) from the root only takes \( O(\log n) \) time by using \( \text{SEARCH} \) in a 2-3 tree. Now consider the path from the root of the tree to \( u \). Clearly, \( A_x \) is the sum of all leaves that are on the left of this path. Therefore, \( A_x \) can be computed by summing up all \( y\.sum \)'s for every node \( y \) that is a left sibling of a node in the path. Since there are only \( \log n \) such nodes \( y \)'s, computing \( A_x \) only takes \( O(\log n) \) time.

Computing \( B_x \) is similar: instead of summing up \( y\.sum \), we sum up \( y\.size \). Therefore, it also takes \( O(\log n) \) time to compute \( B_x \).

Therefore, \( \text{AVERAGE}(x) \) which is \( \frac{A_x}{B_x} \) can be answered in \( O(\log n) \) time.
(b) [3 points] Describe how to modify INSERT to maintain this information. Briefly justify that the worst-case running time for INSERT remains $O(\lg n)$.

**Solution:** Maintaining $x.size$ is similar to what was covered in recitation and homework. Maintaining $x.sum$ is exactly the same: when a node $x$ gets inserted, we simply increase $y.sum$ for every ancestor $y$ of $x$ by the amount $x.key$. Handling overflow for $x.sum$ is exactly the same as $x.size$. Hence, INSERT still runs in worst-case time $O(\lg n)$. 