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 8 double-sided pages, including this one and a double-sided sheet of scratch paper; there should be 12 (numbered) pages of problems.
- This quiz is closed book. You may use one double sided Letter (8.5” × 11”) or A4 crib sheet. No calculators or programmable devices are permitted. Cell phones must be put away.
- 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>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>80</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name: __________________________

Circle your recitation:

<table>
<thead>
<tr>
<th>R01</th>
<th>R02</th>
<th>R03</th>
<th>R04</th>
<th>R05</th>
<th>R06</th>
</tr>
</thead>
<tbody>
<tr>
<td>F10</td>
<td>F11</td>
<td>F12</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
</tr>
<tr>
<td>Joe</td>
<td>Joe</td>
<td>Khanh</td>
<td>Khanh</td>
<td>Emily</td>
<td>Emily</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R07</td>
<td>R08</td>
<td>R09</td>
<td>R10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F11</td>
<td>F12</td>
<td>F1</td>
<td>F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matt</td>
<td>Matt</td>
<td>Geoff</td>
<td>Geoff</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 1. [1 points] Write your name on every page! Don’t forget the cover.

Problem 2. Recurrences [16 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 $T(n) = O(1)$ for $n \leq 2$. For part (e), assume that $T(n, k) = O(1)$ for $n \leq 2$ or $k \leq 2$.

(a) [3 points] $T(n) = 4T(n/4) + \Theta(n \log n)$
   Solution: $T(n) = \Theta(n \log^2 n)$ by case 2 of the (Extended) Master Method.

(b) [3 points] $T(n) = 5T(n/2) + \Theta(n^2 \log n)$
   Solution: $T(n) = \Theta(n^{\log_2 5})$ by case 1 of the Master Method.
(c) [3 points] \( T(n) = 8T(n/2) + n \log n + 2n^3 \)

\[ \text{Solution: } T(n) = \Theta(n^3 \log n) \text{ by case 2 of the Master Method.} \]

(d) [3 points] \( T(n) = 3T(n/4) + n\sqrt{n} \)

\[ \text{Solution: } T(n) = \Theta(n\sqrt{n}) \text{ by case 3 of the Master Method.} \]

(e) [4 points] \( T(n, k) = T(n/2, k) + T(n, k/4) + kn \)

\[ \text{Solution: } T(n, k) = \Theta(kn). \text{ First, use a recursion tree to guess the solution. There are } \log_4(kn) \text{ levels in the tree, and the cost at level } i \text{ is at most } (3/4)^i kn. \text{ The coefficients of } kn \text{ in this expression form a decreasing geometric series whose sum is upper bounded by } 1/(1 - 3/4) = 4. \text{ Thus, the total cost is } O(4kn) = O(kn). \text{ On the other hand, the first level contributes a cost of } kn, \text{ so the total cost is } \Omega(kn). \text{ Use the substitution method to verify that } T(n, k) = 4kn \text{ satisfies the recurrence.} \]
Problem 3. True or False, and Justify [18 points] (6 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] A Monte Carlo algorithm that uses the sequence ‘010101010101...’ (alternating 0 and 1) as its randomness will always run in polynomial time.

Solution: True. A Monte Carlo algorithm always runs in polynomial time regardless of its coins.

(b) T F [3 points] Let $N$ be a positive integer. If there exists an $a$ in $\mathbb{Z}_N^*$ such that $a^2 \equiv 16 \mod N$ and $a \not\equiv 4$ and $a \not\equiv N - 4 \mod N$, then $N$ is composite.

Solution: True. We can rewrite $a^2 \equiv 16 \mod N$ as $(a+4)(a-4) \equiv 0 \mod N$. If $N$ is prime, then for $N$ to divide $(a+4)(a-4)$, $N$ must divide either $a+4$ or $a-4$, so we have $a \equiv 4 \mod N$ or $a \equiv -4 \mod N$. Therefore, if there exists a solution $a \in \mathbb{Z}_N^*$ to $a^2 \equiv 16 \mod N$ that is not equal to 4 or $N - 4$, $N$ must be composite.
(c) T F [3 points] It is possible to devise a Las Vegas algorithm to check whether an $n \times n$ matrix $M$ with binary entries has all 0 entries that runs in expected $O(n)$ time for any input.

Solution: False. An algorithm that is correct on every $n \times n$-bit input must read all $n^2$ bits, so it will take $\Omega(n^2)$ time.

(d) T F [3 points] Searching in a randomized skip list (as presented in class) of $n$ elements never takes $\omega(n)$ time.

Solution: False. A skip list can be of any height, depending on its random choices.
(e) T F  [3 points] Suppose we have computed a minimum spanning tree (MST) and its total weight for some graph $G$. If we make a new graph $G'$ by adding 1 to the weight of every edge in $G$, we will need to spend $\Omega(|E|)$ time to compute an MST and its total weight for the new graph $G'$.

Solution: False. If $T$ is an MST for $G$ with weight $w$, then it is also an MST for $G'$ with weight $w + |V| - 1$.

(f) T F  [3 points] In the amortized analysis of a sequence of $n$ operations using the potential method, the total amortized cost depends on the choice of the potential function.

Solution: True. Different potential functions may yield different amortized costs; there is no unique “amortized cost.” (For example, in the analysis of move-to-front presented in class, we had a different potential function, which gave a different amortized cost, for every other algorithm $A$.)
**Problem 4. Short Answer** [20 points] (5 parts)

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

(a) [3 points] Explain why it is meaningless to say that a function $f(n)$ is “at least $O(n^2)$”.

**Solution:** The statement “$f(n)$ is $O(n^2)$” says that, for large enough $n$, $f(n)$ is upper-bounded by $cn^2$ for some constant $c$. The statement “$f(n)$ is at least $O(n^2)$”, interpreted either as that $f(n)$ is lower-bounded by a function that is upper-bounded by $cn^2$, or that $f(n)$ is upper-bounded by a function that is lower-bounded by $cn^2$, doesn’t give any meaningful information about $f(n)$.

(b) [4 points] In class, we discussed a deterministic linear-time algorithm for finding the median (or the $k$-th smallest element) of an unsorted array. Suppose we changed the algorithm so that it divides the array into groups of size 7, instead of groups of size 5. Give the recurrence for the running time of the resulting algorithm, and state the solution to the recurrence in $\Theta$-notation.

**Solution:** We find the median of each of the groups of size 7, then find the median-of-medians $x$, and partition the array around $x$. Then at least half of the $[n/7]$ groups contribute 4 elements that are greater than $x$, except for possibly one non-full group, and the one group containing $x$ itself. So we have that the number of elements greater than $x$ is at least $4 \left( \left\lfloor \frac{1}{2} \left\lceil \frac{n}{7} \right\rceil \right\rfloor - 2 \right) \geq \frac{4n}{14} - 8 = \frac{2n}{7} - 8$. Thus, in the worst case, the algorithm is called recursively on at most $5n/7 + 8$ elements. Therefore, the running time recurrence is $T(n) \leq T([n/7]) + T(5n/7 + 8) + O(n)$ for $n \geq n_0$ for some constant $n_0$.

We can show that this recurrence solves to $\Theta(n)$ using the substitution method.

(c) [4 points] If you run Johnson’s all-pairs shortest-paths algorithm on a graph with no negative-weight edges, what can you say about the reweighted values \( w_h(u, v) \) compared to the original weight values \( w(u, v) \)?

**Solution:** Since the graph has no negative-weight edges, the shortest path from the new vertex \( s \) to any vertex has weight 0. Thus, \( h(v) = 0 \) for any vertex \( v \). Therefore, \( w_h(u, v) = w(u, v) \).

(d) [4 points] Consider the following two hash functions \( h_1 \) and \( h_2 \) from \( \{1, 2, 3\} \) to \( \{0, 1\} \):

\[
\begin{array}{c|ccc}
  & 1 & 2 & 3 \\
\hline
h_1 & 0 & 0 & 1 \\
\hline
h_2 & 1 & 1 & 0 \\
\end{array}
\]

(i.e., \( h_1(1) = 0, h_1(2) = 0, h_1(3) = 1, h_2(1) = 1, h_2(2) = 1, h_2(3) = 0 \))

1. Is the set \( H = \{h_1, h_2\} \) a universal hash family? Why or why not?
2. What if we modify \( h_1 \) so that \( h_1(2) = 1 \). Is \( H = \{h_1, h_2\} \) a universal hash family now? Why or why not?

**Solution:**

1. \( H \) is not universal, because if \( h \) is selected randomly from \( H \), \( h(1) = h(2) \) with probability 1.

2. If we modify \( h_1 \) so that \( h_1(2) = 1 \), \( H \) becomes universal, since for any pair \( x, y \in \{1, 2, 3\}, x \neq y \), for at most one of the two hash functions do we have \( x \) and \( y \) collide (i.e., the probability that \( h(x) = h(y) \), for \( h \) chosen randomly from \( H \), is at most \( 1/2 = 1/m \), where \( m \) is the number of hash buckets).
(e) [5 points] Define a type 0 array as an array where $3/4$ of the elements are 0 and $1/4$ are 1. Define a type 1 array as an array where $1/4$ of the elements are 0 and $3/4$ are 1. You are given two arrays $A$ and $B$ of length $n$ and are told that one is type 0 and one is type 1, but you do not know which is which.

Design a Monte Carlo algorithm that picks one element uniformly at random from each array and determines which array is which. What is the probability that your algorithm outputs the correct answer? (The probability that your algorithm outputs the correct answer should be greater than 1/2.)

**Solution:** We follow the following procedure:

- Pick an element uniformly at random from each array.
- If the two elements are either both 0 or both 1, flip a coin to determine the answer. Specifically, if the coin is heads, say that $A$ is type-0 and $B$ is type-1; if the coin is tails, say the reverse.
- If the two elements are different, say that the array where we drew the 0 is the type-0 array and the array where we drew the 1 is the type-1 array.

As a result, four things can happen:

- we pick 0's from both arrays,
- we pick 1's from both arrays,
- we pick a 0 from the type-0 array and a 1 from the type-1 array, or
- we pick a 1 from the type-0 array and a 0 from the type-1 array.

The first two cases each happens with probability $(3/4)(1/4) = 3/16$, in which case we guess right with probability $1/2$. The third case happens with probability $(3/4)(3/4) = 9/16$, in which case we answer correctly. The last case happens with probability $(1/4)(1/4) = 1/16$, in which case we answer incorrectly. Thus, overall, the probability we answer correctly is exactly

$$\Pr[\text{correct answer}] = (3/16)(1/2) + (3/16)(1/2) + (9/16) = 12/16 = 3/4.$$ 

A common mistake in the analysis was to assume (implicitly) that $A$ and $B$ are randomly ordered, so that if we get both 0’s or both 1’s, always answering “$A$ is type-0 and $B$ is type-1” is equally good to flipping a coin to guess the answer. The problem is that if $A$, as given to us, is always type-1 (and $B$ is therefore always type-0), this answer would always be wrong. The resulting probability of success would then be $9/16$, which is still better than $1/2$ but not as good as $3/4$.

Another common mistake, and a more serious one, is to confuse conditional probabilities. If you divided your cases according to the result of the choices $a \in A$ and $b \in B$ (i.e., the four cases being $(a, b) = (0, 0), (1, 1), (0, 1), (1, 0)$) instead of according to the result of the choices from whichever array was type-0 and type-1, respectively, you would have to analyze four conditional probabilities of the form

$$\Pr[A \text{ is type-}i, B \text{ is type-}j \mid (a, b) = (i, j)]$$

and not

$$\Pr[(a, b) = (i, j) \mid A \text{ is type-}i, B \text{ is type-}j] = (3/4)(3/4) = 9/16.$$ 

It is still possible to carry out this analysis using Bayes’ rule, but it is much messier than the solution above, which avoids conditional probabilities altogether.
Solution: (Alternative solution:) An even simpler solution is to take just the element from $A$ (ignoring $B$), and if it is a 0 to say “$A$ is type-0 and $B$ is type-1,” and if it is a 1 to say “$A$ is type-1 and $B$ is type-0.” To analyze this, suppose first that $A$ is in fact type-0. Then we draw a 0 from $A$ with probability $3/4$, in which case we answer correctly. Alternatively, if $A$ is in fact type-1, then we draw a 1 from $A$ (and therefore answer correctly) with probability again $3/4$. Although the two “events” of $A$ being type-0 or type-1 cannot be assigned probabilities (we have no idea how $A$ and $B$ are given to us), in either case we answer correctly with probability $3/4$, so overall we answer correctly with probability $3/4$ as well.
Problem 5. Postering the Corridor [12 points] (3 parts)

A certain university has a very long corridor with \( n \) boards (labeled 1 through \( n \)) where people can place posters. Due to university rules, you may put up as many posters advertising an event as you want, provided that:

- The same poster can appear only once per board, and
- The same poster cannot appear on two adjacent boards. That is, you are prohibited from placing a poster on both board \( i \) and board \( i + 1 \).

(Assume that there is always space to put up a poster on any board.)

Through extensive spy-camera research, you know that each person only looks at one of the boards. Furthermore, you can predict that exactly \( b_i \) people will look at board \( i \).

Imagine you have an event you wish to publicize, and you want to select which boards to put posters on in order to maximize the number of viewers while obeying university rules. Thus, you want to design an algorithm that, given \( \{b_1, b_2, \ldots, b_n\} \), outputs a set of board numbers \( S \) that maximizes \( \sum_{i \in S} b_i \), subject to the constraint that if \( i \in S \), then \( (i + 1) \not\in S \) and \( (i - 1) \not\in S \).

(a) [3 points] Consider the following greedy algorithm for placing posters to advertise an event:

- Define a still-available board as a board that you have not put a poster on and that is not adjacent to one you have put a poster on.
- While there are any still-available boards remaining:
  - Select the still-available board with the greatest number of viewers, and put a poster on it.

Give an example (i.e. a configuration of \( b_i \) values) that shows that this algorithm does not always give the arrangement that maximizes the number of viewers.

Solution: One possible example is \( b_1 = 2, b_2 = 3, b_3 = 2 \). The greedy algorithm will select board 2 only, and will therefore get 3 viewers; the optimal solution is to select boards 1 and 3 for a total of 4 viewers.
(b) [6 points] Give an efficient algorithm that uses dynamic programming to find the maximum possible number of viewers of a poster that obeys the rules.

Solution: Define $f(i)$ as the maximal number if we only consider putting posters on boards 1 through $i$. Thus, $f(0) = 0$, $f(1) = b_1$, and $f(n)$ is the solution to the overall problem we are seeking.

We now consider the problem of calculating $f(i)$. If we put a poster on board $i$, we get the value for $b_i$ but cannot use board $(i - 1)$. Alternatively, if we do not, we can use any of the previous boards. We thus get the recurrence

$$f(i) = \max(f(i - 1), b_i + f(i - 2)).$$

We can solve this recurrence through dynamic programming, either by starting up from $i = 2$ (the first value of $i$ greater than our base cases) or through recursion with memoization, e.g.:

1. $f[0] = 0$
2. $f[1] = b_1$
3. for $i = 2$ to $n$
4. $f[i] = \max(f[i - 1], b_i + f[i - 2])$
5. return $f[n]$

(c) [3 points] In a few sentences, explain how you would modify your algorithm from part (b) to return not only the maximum number of viewers itself, but a configuration of boards obeying the rules that gives that maximum.

Solution: Have each call of the algorithm return not only the maximum for that sub-problem, but also a set $s(i)$ that yields that maximum. Whenever we calculate

$$f(i) = \max(f(i - 1), b_i + f(i - 2)),$$

set $s(i) = s(i - 1)$ if $f(i - 1)$ was maximal and $s(i) = s(i - 2) \cup \{i\}$ otherwise. (For the base cases, $s(0) = \{\}$, $s(1) = \{1\}$.)
Problem 6. Selection in B-Trees [13 points] (3 parts)

Recall that a B-tree of minimum degree \( t \geq 2 \) is a search tree having the following properties:

- Every node \( x \) has an attribute \( x.n \) (the number of keys stored in \( x \)), and \( x.n \) keys \( x.key_1, \ldots, x.key_n \) (in increasing order).
- Every internal node \( x \) contains \( x.n + 1 \) pointers \( x.c_1, \ldots, x.c_{x.n+1} \) to its children.
- The keys \( x.key_i \) separate the ranges of keys stored in each subtree, i.e., if \( k_i \) is any key stored in the subtree rooted at \( x.c_i \), then \( k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1} \).
- Every non-root node contains between \( t - 1 \) and \( 2t - 1 \) keys.
- Every leaf in the B-tree occurs at the same depth.

B-trees support efficient SEARCH, INSERT, and DELETE operations.

The rank of a key \( v \) in a B-tree \( T \) is the number of keys in \( T \) that are less than or equal to \( v \). We will assume that all keys in the B-tree are distinct.

Suppose we wish to augment B-trees to support a RANK operation, which determines the rank of a given key in the tree, and a SELECT operation, which determines the key of a given rank in the tree.

(a) [5 points] Suppose every node \( x \) of a B-tree contains an attribute \( x.size \), which gives the total number of keys in the subtree rooted at \( x \), including the keys in \( x \). Given such a B-tree, give an algorithm \( \text{RANK}(x, v) \) that finds the rank of \( v \) in the subtree rooted at \( x \). (You may assume that the key \( v \) exists in the subtree rooted at \( x \).) Your algorithm should run in \( O(t \log_t n) \) time.

Solution:

\[
\begin{align*}
\text{RANK}(x, v) & \quad \text{if } x \text{ is a leaf} \\
& \quad \text{for } i = 1 \text{ to } x.n \\
& \quad \quad \text{if } v == x.key_i \\
& \quad \quad \quad \text{return } i \\
& \quad \quad \text{rank} = 0 \\
& \quad \quad i = 1 \\
& \quad \text{while } i \leq x.n \text{ and } v > x.key_i \\
& \quad \quad \text{rank} = \text{rank} + x.c_i.size + 1 \\
& \quad \quad i = i + 1 \\
& \quad \text{if } i \leq x.n \text{ and } v == x.key_i \\
& \quad \quad \text{return } \text{rank} + x.c_i.size + 1 \\
& \quad \text{else return } \text{rank} + \text{RANK}(x.c_i, v)
\end{align*}
\]
(b) [5 points] Suppose again that every node $x$ of a B-tree contains an attribute $x.size$, which gives the total number of keys in the subtree rooted at $x$, including the keys in $x$. Given such a B-tree, give an algorithm $\text{SELECT}(x, i)$ that finds the $i$-th smallest key in the subtree rooted at $x$. (You may assume that the subtree rooted at $x$ contains at least $i$ keys.) Your algorithm should run in $O(t \log_t n)$ time.

Solution:

\begin{algorithm}
\text{SELECT}(x, i) \\
1 \hspace{1em} \textbf{if} \ x \ \text{is a leaf} \\
2 \hspace{2em} \textbf{return} \ x.key_i \\
3 \hspace{1em} \textbf{for} \ j = 1 \ \text{to} \ x.n \\
4 \hspace{2em} \text{rank} = x.c_j.size + 1 \\
5 \hspace{2em} \textbf{if} \ \text{rank} == i \\
6 \hspace{3em} \textbf{return} \ x.key_j \\
7 \hspace{2em} \textbf{else} \ \textbf{if} \ i < \text{rank} \\
8 \hspace{3em} \textbf{return} \ \text{SELECT}(x.c_j, i) \\
9 \hspace{2em} \textbf{else} \ i = i - \text{rank} \\
10 \hspace{1em} \textbf{return} \ \text{SELECT}(x.c_{x.n+1}, i)
\end{algorithm}

(c) [3 points] Describe how to implement an INSERT procedure that efficiently maintains a $size$ attribute for each node. Your algorithm should run in $O(t \log_t n)$ time.

Be sure to handle the case where INSERT splits one or more nodes. It is sufficient to describe how to modify the INSERT procedure for B-trees that is described in the book.

Solution: We modify the INSERT procedure from the book, as follows.
For every node $x$ on the path from the root to the leaf where we insert the key, increment $x.size$ by 1.
Whenever we split a full node, for each of the two resulting nonfull children $x$, compute $x.size$ as $1 + \sum_{i=1}^{x.n+1} x.c_i.size$. The size of the parent node does not change, unless the node we have split is the root, in which case we compute $x.size$ for the new root $x$ as $1 + x.c_1.size + x.c_2.size$. 
SCRATCH PAPER