Quiz 1 Solutions

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains 5 problems, several with multiple parts. You have 80 minutes to earn 80 points.
- This quiz booklet contains 13 pages, including this one, and a sheet of scratch paper.
- This quiz is closed book. You may use one double-sided letter (8\frac{1}{2}" × 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. If you run out of space, continue your answer on the back of the same sheet and make a notation on the front of the sheet.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to “give an algorithm” in this quiz, describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- Do not spend too much time on any one problem. Generally, a problem’s point value is an indication of how many minutes 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. Please be neat.
- Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Title</th>
<th>Points</th>
<th>Parts</th>
<th>Grade</th>
<th>Initials</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Name</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>True or False</td>
<td>20</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Short Answers</td>
<td>24</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Discovering Fakes</td>
<td>10</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Finding Repetitions</td>
<td>10</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Dynamic Quiz Takers</td>
<td>15</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name: ____________________________

Circle your recitation:

F10  F11  F11  F12  F12  F1  F2  F3
R01  R02  R07  R03  R08  R04  R05  R06
Yotam Boon Teik Aizana Annie Aizana Annie Katherine Heejung
Problem 0. Name. [1 points] Write your name on every page of this exam booklet! Don’t forget the cover.

Problem 1. True or False. [20 points] (10 parts)
Circle T or F for each of the following statements to indicate whether the statement is true or false and briefly explain why.

(a) T F  Given $n$ numbers, the worst-case running time of the median finding algorithm is $\Theta(n \log n)$.

Solution: [2 points] False. The running time of the median finding algorithm is $O(n)$.

(b) T F  On the same input on different executions, the multivariate polynomial identity testing algorithm (from recitation 3) may produce different answers.

Solution: [2 points] True. If two polynomials are not identical, then it’s possible to get both true and false answers when executing the test.

(c) T F  In the worst case, the running time for a search in a skip list is $\Theta(n)$.

Solution: [2 points] True. $O(\log n)$ was expected, but in the worst case, we may have to search through $\Theta(n)$ elements. Recall that upon an insertion of an element, promotion of the element to the upper levels is done randomly. In the worst case, it’s possible that no elements get promoted or all elements get promoted to the upper levels.

(d) T F  Given a set of points $\{x_1, \ldots, x_n\}$, and a degree $n$ bounded polynomial $P(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$, we can evaluate $P(x)$ at $\{x_1, \ldots, x_n\}$ in $O(n \log n)$ time using the FFT algorithm presented in class.

Solution: [2 points] False. Using FFT, we can efficiently evaluate a polynomial at the roots of unity, but not any $n$ points.
(e) T F If we augment the “paranoid” quicksort (from lecture) to only pick each potential pivot once, the worst-case running time of “paranoid” quicksort is worse than that of the randomized quicksort.

**Solution:** [2 points] False. The worst-case running time of the randomized quicksort is $\Theta(n^2)$, and the augmented “paranoid” quicksort takes $O(n^2)$ time since each pivot takes $O(n)$ to check and there are $O(n)$ bad pivots, so the recursion for “paranoid” quicksort becomes $T(n) = T(n/4) + T(3n/4) + O(n^2)$. Using induction, one can show that $T(n) \leq cn^2$ for some constant $c$. Thus, it is not worse.

(f) T F In a weighted connected graph $G = (V,E)$, each edge with the minimum weight belongs to some minimum spanning tree of $G$.

**Solution:** [2 points] True. Consider a MST $T$. If it doesn’t contain the minimum edge $e$, then by adding $e$ to $T$, we get a cycle. By removing a different edge than $e$ from the cycle, we get a spanning tree $T'$ whose total weight is no more than the weight of $T$. Thus, $T'$ is a MST that contains $e$.

(g) T F In a weighted connected graph $G$, if $s$ is a starting node in Prim’s algorithm, then for any other vertex $v$, the path on the resulting MST from $s$ to $v$ is the shortest path.

**Solution:** [2 points] False. Consider graph $G$ with vertices $\{s, v, w\}$ and edge weights $e(s,v) = 3$, $e(s,w) = 2$ and $e(w,v) = 2$. Path on the MST from $s$ to $v$ is of cost 4, while the shortest path is 3. There are many examples possible.
(h) **T F** If

\[
T(n) = \begin{cases} 
T(an) + T(bn), & \text{if } n > n_0 \\
\ c, & \text{otherwise}
\end{cases}
\]

where \( a, b > 0 \) and \( a + b < 1 \), and \( c \) and \( n_0 \) are some constants, then \( T(n) \) is \( O(n) \).

**Solution:** [2 points] True. Can be proven by simple induction.

---

(i) **T F** For a given weighted connected graph \( G = (V, E) \), we would like to find the longest simple path between any two vertices. We can solve this problem by negating the edge weights and running Johnson’s algorithm.

**Solution:** [2 points] False. Negation of edge weights may create a negative cycle.

---

(j) **T F** In a weighted graph \( G \), if \( k \) is the maximum number of edges in shortest paths between any two vertices, then it is possible to reduce the running time of Floyd-Warshall to \( O(kn^2) \) by finishing early.

**Solution:** [2 points] False. Floyd-Warshall iterates over a ordered list of vertices \( (v_1, \ldots, v_n) \), and if the shortest path with length \( k \) or less between two vertices uses node \( v_n \), it will not find the shortest path until the last loop.
Problem 2. Short Answers. [24 points] (6 parts)

Please answer the following questions.

(a) Give asymptotic upper and lower bounds for $T(n)$ in the following recurrences. Make your bounds as tight as possible. You can assume that $T(n)$ is constant for $n \leq 2$.

$$T(n) = T(n^{1/3}) + T(n^{2/3}) + \log n$$

Solution: [4 points] Do a change of variable using $m = \log n$. You will end up with $T(2^m) = T(2^{m/3}) + T(2^{2m/3}) + m$. If we define $S(m) = T(2^m)$, we will get $S(m) = S(m/3) + S(2m/3) + m$. By drawing the recursion tree, we get $S(m) = \Theta(m \log m)$. Changing the variable back gives us $T(n) = \Theta(\log n \log \log n)$. It is also possible to solve this by drawing the recurrence tree for $\log n$ where there are $O(\log \log n)$ levels and each level has $\log n$ work.

(b) We would like to test whether $A \times B = C$, where $A$, $B$, and $C$ are $n \times n$ matrices. Suppose it takes $O(k)$ to generate a random bit. Give an algorithm for the matrix identity test such that the error rate is less than $\epsilon$. What is the running time of your algorithm?

Solution: [4 points] If $A \cdot B \neq C$, Freivald’s Algorithm has an error rate of less than $1/2$. Otherwise, it is always correct. Hence, the error rate will be less than $2^{-x}$ if we run the algorithm for $x$ times and output “equal” if the algorithm returns “equal” every time. Setting $2^{-x} = \epsilon$, we get $x = \log \frac{1}{\epsilon}$. Each time we run, we need to generate $n$ random bits. Therefore, the total running time is $O((nk + n^2) \log \frac{1}{\epsilon})$. 

(c) Prove or disprove using a counterexample: Let $G$ be a flow network graph. If all the capacities of $G$ have unique integer capacities, then there is a unique maximum flow.

**Solution:** [4 points] This statement is false. Consider the graph given below:

On this graph, the maximum flow is 1 because of the capacity on the edge $(v, t)$. However, it is possible to achieve this flow either by sending one unit through the path $(s, v, t)$ or to send one unit along $(s, u, v, t)$. Therefore, the flow is not unique even though the capacities all are.
(d) Draw the residual graph $G_f$ given the graph $G$ and the flow $f$ below. The notation $x/y$ means that there is currently $x$ flow going through an edge of capacity $y$.

**Solution:** [4 points] The residual graph is given below.
(e) For a set $A$ of integers in the range $[0...100n]$, give an $O(n \log n)$ algorithm which finds whether there is a triple of three (not necessarily distinct) elements $(x, y, z)$ in $A$ such that $z = x + y + c$ where $c \in [-\log n, ..., +\log n]$. Justify the runtime.

**Solution:** [4 points] This problem is similar to the problem on homework. Create a polynomial $P(x)$ from the set:

$$P(x) = x^{a_1} + x^{a_2} + ...x^{|A|}$$

where $a_i \in A$. Square the polynomial, which can be done efficiently using FFT, to obtain a new polynomial

$$Q(x) \equiv P(x)^2 = q_0x^0 + q_1x^1 + ... + q_{200n}x^{200n}$$

Then $q_k$ will be exactly the number of pairs $i, j$ such that $a_i + a_j = k$ (note that $q_k$ might be zero). The first method then compares the first $100n$ resulting positive coefficient values against our original set to see if these match. This can be done by hashing the resulting values, and then iterating over the original set to test for inclusion. The same process is used to check for the $+i$ and $-i$ offset for all $i \in [-\log n, ..., +\log n]$. The second method notes that $Q(x)$ can be multiplied by the polynomial $x^{-\log n} + x^{-\log n+1} + ... + x^{-1} + 1 + x + ... + x^{\log n}$ using FFT and the powers of $x$ that have nonzero coefficients in the result can be compared to $A$ in time $O(n)$.

(f) You would like to design an algorithm which has the following specification: Given four polynomials $A, B, C, D$, each of degree at most $n$, if $AB = CD$, output PASS with probability 1. If $AB \neq CD$, output FAIL with probability at least $1 - 1/\log n$.

Your algorithm does the following once: chooses uniformly at random an integer $x$ from a set of integers $\{1, ..., m\}$ and checks if $A(x) \times B(x) = C(x) \times D(x)$. If the answer is yes, the algorithm outputs PASS, otherwise the algorithm outputs FAIL.

How big should $m$ be in order to satisfy the specification?

**Solution:** [4 points] $m$ should be $2n \log n$. If $AB = CD$, then the algorithm always outputs PASS. Otherwise, $AB \neq CD$. The product of two degree $\leq n$ polynomials has degree at most $2n$, so the probability that for random $x$, $A(x) \times B(x) = C(x) \times D(x)$ is at most $2n/m$ which by our choice of $m$ is at most $1/\log n$. 
Problem 3. Finding Fake Coins [10 points] (2 parts)

You are given a bag of coins, most of which have same weight. There is a possibility that some fake coins (which have different weight) are mixed into the bag. You want to find the fake coins or make sure that all coins in the bag are real using a scale that can compare the weight of a set of coins to another set of coins. (There is no way to measure absolute weights of coins.)

For the following settings, give the most efficient deterministic algorithm to find the fake coin(s) or show that there are none. Write a recurrence for your algorithm and solve the recurrence.

(a) A bag of $n$ coins may contain up to 1 fake coin that is heavier than real coins. Find the fake coin if it is in the pile.

Solution: Divide the coins into 2 piles. Compare the two piles. If one pile is heavier, recurse on that pile. If they are the same, there is no fake coin.

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) = \Theta(\log n)$$

Alternative Solution: Divide the coins into 3 piles. Choose two piles and compare them. If one pile is heavier, recurse on it. If they are even, the fake coin may be in the remaining pile or does not exist. Recurse on the pile you didn’t compare.

$$T(n) = T\left(\frac{n}{3}\right) + \Theta(1) = \Theta(\log n)$$

(b) A bag of $n$ coins may contain up to 2 heavy fake coins. Find both fake coins in the pile if there are 2 or find one if there is 1.

Solution: [5 points] Divide the coins into 2 piles and compare them.

If the comparison is even, it means that (1) two piles each have one heavy coin or (2) there are no fake coins. Since there is at most one heavy coin in each pile, run algorithm from (a) on each pile to discover the fake coin in each pile. If the comparison is uneven, it means the heavy pile has all the fake coins. Recurse on the heavy pile.

Base case, you have two piles of one coin each. If one is heavier, it is a fake. If they are the same, you will have to test against a third coin.

The recursion is

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + \Theta(1), & \text{if piles do not weigh the same} \\ \Theta(\log n), & \text{if piles weigh the same.} \end{cases}$$

Both cases have $\Theta(\log n)$ run time, so $T(n) = \Theta(\log n)$.

Alternative Solution: Divide the coins into 3 piles and compare them all there.

If they all weigh the same, then there are no fake coins. If two piles are heavier, combine these two piles and recurse on the combined pile. If one pile is heavier, there is one or two fake coins in it. Recurse on this pile. In the base case, you have 3 piles of 1 coin each. Weigh them against each other. The heavy two are the fake coins.

The recursion is

$$T(n) = T\left(\frac{2n}{3}\right) + \Theta(1) = \Theta(\log n).$$
Problem 4. Finding Repetitions [10 points] (2 parts)

(a) [4 points] Suppose there exists an integer that appears more than $n/2$ times in the array. Give a linear-time deterministic algorithm to find such an integer.

Solution: Use the deterministic linear-time median-finding algorithm covered in lecture to find the median of the array – it will be the majority element. Since, if the median wouldn’t be the majority element, then the majority element would be smaller or larger than the median, which is impossible (e.g., there are only $n/2$ elements smaller than the median).

Note that hashing is not deterministic linear-time. Answers that used hashing received partial credit. One can imagine using an auxiliary array instead of a hash table and indexing into the array with the integers in the given array, but the size of this auxiliary array is unbounded.

(b) [6 points] Let $k$ be a given integer constant where $k > 2$. Now suppose there exists an integer that appears more than $n/k$ times (you can assume $n$ is divisible by $k$). Give a linear-time deterministic algorithm to find all such integers.

Solution: Let’s call an integer that appears more than $n/k$ times a common element. Select elements $a_1, a_2, \ldots, a_k$, which are the elements with ranks $\frac{n}{k}, \frac{2n}{k}, \ldots, \frac{(k-1)n}{k}$, respectively. The $k$ selections each takes $O(n)$ time. Then check each of the elements $a_1, a_2, \ldots, a_k$, to see whether it is a common element. Each of the $k$ checks take $O(n)$ time. The overall runtime is therefore $O(nk)$.

The correctness of the algorithm follows from the claim that any common element must be among $a_1, a_2, \ldots, a_k$. To see that, consider the sorted array. Any common element is a contiguous block of size $> n/k$. Since our “probes” $a_1, a_2, \ldots, a_k$ are at distance $n/k$, they must strand the block of a common element.

See paragraph about hashing in the answer for part (a).
Problem 5. Dynamic Quiz Takers [15 points] (3 parts)

In a fictional class, every student takes a stance in each quiz – either to work hard (W), or to slack off (S). 50% of the hard-workers receive an A for the quiz, 40% a B, 10% a C, while for slackers, only 10% receive an A, 20% a B, 30% a C, and 40% fail the quiz. Moreover, 20% of the hard-working students in one quiz become slackers in the next quiz, while 30% of the slackers in one quiz become hard-workers in the next quiz, independent of all previous quizzes. This model is summarized by the following diagram. The class begins with 80% hard-workers and 20% slackers, and has a series of \( n \) quizzes.

\[ \begin{align*}
& \text{(a)} \quad \text{[4 points] What is the proportion of students that receive } (A,F) \text{ for their first two quizzes, in order? Given a student with grade history } (A,F) \text{, what stances did the student most likely take (i.e., what is the stance history of the largest proportion, among students with such grade history)?} \\
& \text{Fact: Given events } U, A, B, X \text{ such that } A \cup B = U \text{ and } A \cap B = \phi, \text{ } \\
& \quad \Pr(X|U) = \Pr(X|A) \Pr(A|U) + \Pr(X|B) \Pr(B|U). \\
& \text{Solution: } 3.76\%, (W,S). \\
& \Pr((A,F)) = \Pr((A,F) | (W,S)) + \Pr((A,F) | (S,S)) \\
& \quad = \Pr(W \text{ for quiz 1}) \times \Pr(A|W) \times \Pr(S \text{ for quiz 2} | W \text{ in quiz 1}) \times \Pr(F|S) \\
& \quad \quad + \Pr(S \text{ for quiz 1}) \times \Pr(A|S) \times \Pr(S \text{ for quiz 2} | S \text{ in quiz 1}) \times \Pr(F|S) \\
& \quad = 0.8 \times 0.5 \times 0.4 \times 0.1 \times 0.7 \times 0.4 \\
& \quad \quad = 0.032 + 0.0056 = 0.0376 \\
& \text{The proportion of students with grade history } (A,F) \text{ whose stances were } (W,S), \text{ which is } 0.8 \times 0.5 \times 0.2 \times 0.4, \text{ is larger than the proportion of students whose stances were } (S,S), \text{ which is } 0.2 \times 0.1 \times 0.7 \times 0.4. \text{ Stance histories } (W,W) \text{ and } (S,W) \text{ do not generate such a grade history. This means the students who received } (A,F) \text{ most likely had the stance history } (W,S). \end{align*} \]
(b) [8 points] Given the grade history of a student, \( G_n = (g_1, \ldots, g_n) \), where \( g_i \in \{A, B, C, F\} \) for \( i = 1, \ldots, n \), we wish to determine the most probable stance history of the student, \( H(G_n) = (h_1, \ldots, h_n) \), where \( h_i \in \{W, S\} \) for \( i = 1, \ldots, n \).

For example, if \( n = 2 \), given \( G_2 = (A, A) \), the algorithm should return \( H(G_2) = (W, W) \); given \( G_2 = (A, F) \), the algorithm should return your answer in part (a).

Devise an algorithm that computes this using dynamic programming.

**Solution:** We memoize the proportion of the most probable stance history for each subproblem \( G_k = (g_1, \ldots, g_k), k = 1, \ldots, n \). Let \( V(G_k, h_k) \) denote the proportion of students in the most probable stance history with stance \( h_k \) in quiz \( k \). Then,

\[
V(G_k, W) = \Pr[g_k | W] \times \max\{.8 \times V(G_{k-1}, W), .3 \times V(G_{k-1}, S)\}
\]

and

\[
V(G_k, S) = \Pr[g_k | S] \times \max\{.2 \times V(G_{k-1}, W), .7 \times V(G_{k-1}, S)\}.
\]

The base cases, \( V(g_1, W) \) and \( V(g_1, S) \) where \( g_1 \in \{A, B, C, F\} \), are given in the previous diagram. While constructing the table, record the maximum argument on the right-hand side of the equation above to allow for backtracking. The final solution is determined by finding the larger of \( V(G_n, W) \) and \( V(G_n, S) \), and then backtracking to give the most probable preceding stances.

(c) [3 points] What is the runtime complexity of your algorithm, in terms of \( n \)? Compare this with the complexity of the brute force approach (i.e., the solution that iterates through all possible stance histories) to demonstrate the advantage of dynamic programming.

**Solution:** The running time of the algorithm is \( O(n) \), which is significantly more efficient than that of the brute-force solution, \( O(2^n) \).