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 7 numbered pages of problems.
- This quiz is closed book. You may use one double sided Letter ($8\frac{1}{2}$” $\times$ 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 covered in lectures, homework or recitations. 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>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>80</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name: ____________________________

Circle your recitation:

- R01
- R02
- R03
- R04
- R05
- R06
- F10
- F11
- F12
- F1
- F2
- F3
- Rafael
- Shaunak
- Lin
- Lin
- Yu
- Yu
- R07
- R08
- R09
- R10
- F11
- F12
- F1
- F2
- Piotr
- Piotr
- Tom
- Tom
Problem 1. [1 points] Write your name on every page! Don’t forget the cover.

Problem 2. True or False, and Justify [12 points] (4 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 A dynamic program’s sub-problem graph (where there is an edge from sub-problem A to sub-problem B iff solving A requires a solution to B) must be a tree.

**Justification:**

**Solution:** False. The subproblem graph is directed. As many students noted, this graph cannot have any cycles: if there was a cycle in the graph, then no subproblem in the cycle can ever be solved first because it depends on other subproblems in the cycle. However, a directed acyclic graph is not necessarily a tree. As a simple counterexample, consider the graph with vertices $A, B$ and $C$ and with edges from $(A, B), (A, C)$ and $(B, C)$.

(b) T F Let $h$ be a function chosen uniformly at random from a universal hash family that maps keys from the domain $\{1, \ldots, n\}$ to the range $\{1, \ldots, m\}$. Let $a, b, c \in \{1, \ldots, n\}$ such that $a \neq b, a \neq c$ and $b \neq c$.

The probability that $h(a) = h(b) = h(c)$ is at most $1/m$.

**Justification:**

**Solution:** True. The probability that $h(a) = h(b) = h(c)$ is bounded by the probability that $h(a) = h(b)$, which is at most $1/m$ because $h$ is drawn from a universal family.

Note that the probability that $h(a) = h(b)$ is at most $1/m$, and that the probability that $h(b) = h(c)$ is at most $1/m$, but these facts do not imply that the probability that $h(a) = h(b) = h(c)$ is at most $1/m^2$. The two events are not necessarily independent, so the probabilities do not multiply. Giving this incorrect justification is not enough to solve the problem.
(c) T F You are given a weighted graph $G = (V, E, W)$ such that (i) the weights $W$ are non-negative and (ii) $|E| = |V|^{1.5}$. Your goal is to compute all-pairs shortest paths in $G$.

Among all algorithms seen in the class, Floyd-Warshall algorithm achieves the lowest running time for this problem.

Justification:

Solution: False. The Floyd-Warshall algorithm takes $O(|V|^3)$ time to solve this problem. However, since the edges of the graph have non-negative weights, we can solve all-pairs shortest paths by running Dijkstra’s algorithm once from each vertex. The total running time of this algorithm is $O(|V||E| + |V|^2 \log |V|)$, which is $O(|V|^{2.5})$ for this particular problem. Note that running Dijkstra’s once is insufficient, since it only computes single-source shortest paths.

(d) T F Assume that we we have a randomized primality testing algorithm that given a number $N$ does the following (i) if the number is prime, it always reports YES and (ii) if the number is composite, it outputs NO with probability at least $3/4$.

In order to ensure that the algorithm reports NO in case (ii) with probability at least $1 - 1/k$, we need to repeat it $k$ times.

Justification:

Solution: False. The probability that the algorithm returns NO in a single trial is $3/4$, so the probability that the algorithm returns NO on any of $t$ trials is $1$ minus the probability that the algorithm returns YES on all $t$ trials, which is $1 - (1/4)^t$.

To make this probability at least $1 - 1/k$, we only need to run $t = \log_4 k$ trials, or $O(\log k)$ trials - much less than $k$ trials.
Problem 3. Short Answer [20 points] (4 parts)

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

(a) Solve the recurrence $T(n) = 8T(n/2) + n^3 \lg n$, i.e. provide a function $f(n)$ such that $T(n) = \Theta(f(n))$. As usual, assume that $T(n) = O(1)$ for $n \leq 2$.

Solution: Case 2 of the master theorem (as seen in Lecture 2) states that if $T(n) = aT(n/b) + \Theta(n^p \log^q n)$ and $p = \log_b a$ then $T(n) = n^p \log^{q+1} n$. In this case $p = 3 = \log_2 8 = \log_b a$ so case 2 applies. $q = 1$, thus $T(n) = \Theta(n^3 \log^2 n)$.

Some students attempted to use case 3, however this requires that the work done at the root is polynomially larger than the number of leaves ($p \log_b a$) which is not the case here.

(b) You are given a set $P$ of $n$ points in the plane $(x_i, y_i)$ for $i = 1, 2, ..., n$. Give an efficient randomized algorithm for determining if there exist three points in $P$ that are co-linear, i.e., lie on the same line.

For full credit your algorithm should run in expected time $O(n^2)$. You can assume that all arithmetic is exact.

Solution: Compute the line $m_i x + b_i$ through each pair of points. Observe that there are two duplicate lines ($m_i = m_j$ and $b_i = b_j$ for $i \neq j$) if and only if there are at least three co-linear points. We can identify if there are two such duplicate lines by hashing all $(m_i, b_i)$ pairs and checking the collisions for duplicates in $O(n^2)$ expected time. Alternatively, we can find duplicates by sorting the $(m_i, b_i)$ pairs in $O(n^2 \log n)$ time and scanning through the sorted list checking if adjacent pairs are equal (note that this solution was given slightly less credit).

There were two common errors on this problem. One was to only check the slopes for duplicates which incorrectly counts parallel lines as co-linear. The second was to try and perform some kind of random sampling of the vertices and then only check those vertices. For example, some students randomly selected a vertex and checked every other pair of points to see if the three points formed a line. These methods all succeed with very low probability. In the case of the example, if there is only one set of three co-linear points the probability of finding it is $3/n$. 
(c) Give an efficient randomized Monte Carlo algorithm that, given two \(n \times n\) matrices \(A\) and \(B\), checks whether \(AB = BA\), i.e., whether the matrices commute.

**Solution:** We will directly apply Freivald’s matrix product checking algorithm from class. Select an \(n \times 1\) vector \(x\) with each element drawn uniformly at random from \(\{0, 1\}\). Compute \(A(Bx)\) and check if it equals \(B(Ax)\). If they are equal, output that the matrices commute, otherwise output that they do not. If the matrices do commute, then for all \(x\) we have \(A(Bx) = B(Ax)\) so in this case our algorithm always outputs correctly. If the matrices do not commute, then by the analysis given in class we have \(A(Bx) \neq B(Ax)\) with probability at least \(1/2\). Since multiply an \(n \times n\) matrix by an \(n \times 1\) vector takes \(O(n^2)\), and we do this a constant number of times, the total running time is \(O(n^2)\).

One common solution attempt for this problem was to sample a random row and column from matrix products \(AB\) and \(BA\) and only compare those. Observe that if the only difference between \(AB\) and \(BA\) was one entry, then this solution only outputs the correct answer with probability \(1/n^2\).

(d) Recall that a heap performs insertion and deletion in worst case \(O(\log n)\). Argue that (without changing the data structure), a heap can perform insertion in amortized \(O(\log n)\) and deletion in amortized \(O(1)\).

**Solution:** The basic intuition here is that we can pay for the deletions by charging extra for the insertions (\(2\log n\) instead of \(\log n\), since each item can only be deleted once. It is not quite correct to say that we can pay for the deletion of an item \(x\) at the time of its insertion, as when it is inserted \(n\) may be much smaller than when it is deleted. Instead, a deletion is paid for by the last insertion before it that hasn’t already paid for a deletion. The easiest way to formalize this is with the potential method.

For our potential function, we will use \(\Phi(D) = \sum_{i=1}^{n} \log i = \log i!\). The change in potential caused by an insertion is \(\Delta \Phi(D) = \log n\). Thus the amortized cost of an insertion is \(\log n + \log n = O(\log n)\). The change in potential caused by a deletion is \(\Delta \Phi(D) = -\log n\) thus the amortized cost of a deletion is \(\log n - \log n = O(1)\) as desired.

Many students attempted to reason about the specific structure of heaps, but observe that this logic applies to any data structure with insertion and deletion operations. If insertion takes \(O(f(n))\) and deletion takes \(O(g(n))\) then we can show that insertion takes amortized \(O(f(n) + g(n))\) and deletion takes amortized \(O(1)\). Some students also attempted to restrict the sequence of operations including deletes, however the amortized cost of deletion must hold regardless of the way deletions are interleaved with insertions.
Problem 4. A tale of two programs [12 points]

In this problem we assume that all arithmetic operations are modulo 2.

You are given an access to two programs $P_1$ and $P_2$. Both $P_1$ and $P_2$ take a binary vector $x$ of length $n$ as an input, and produce a binary vector of length $n$ as an output. Both of them are supposed to compute $A_1x$ for some binary $n \times n$ matrix $A_1$. Alas, only one of them ($P_1$) does it correctly; the other program ($P_2$) computes $A_2x$, where $A_1$ and $A_2$ differ in one entry.

Show how to identify the entry on which $A_1$ and $A_2$ differ using a small number of calls to programs $P_1$ and $P_2$. Note that you do not have any direct access to the matrices $A_1$ and $A_2$.

**Solution:** We start by querying $P_1$ and $P_2$ with $x = [1, 1, 1, \ldots, 1]$. $P_1x$ and $P_2x$ differ at only one position $i$ which is the row that the different entry lies in. The column of the different entry can be found via binary search. The algorithm starts with a half 0’s half 1’s vector $x_1 = [1, 1, \ldots, 1, 0, 0, \ldots 0]$. If $P_1x_1 = P_2x_1$, the different entry must lie in the second half of the positions. The second half is then split equally into 0’s and 1’s, so $x_2 = [0, 0, \ldots, 0, 1, 1, \ldots, 1, 0, 0, \ldots 0]$ with $n/4$ 1’s is queried. If $P_1x_1 \neq P_2x_2$, the different entry must lie in the first half of the positions. Then we split the first half into 0’s and 1’s and query $x_2 = [1, 1, \ldots, 1, 0, 0, \ldots 0]$ with $n/4$ 1’s. By repeating this process, we will end up with only one nonzero position $j$, which is the column of the different entry. The position of the different entry is therefore $(i, j)$. The algorithm takes $O(\log n)$ running time.

Many students showed how to find the row of the entry, but not the column. To find the column, some students used an algorithm that queries $x_i$, $i = 1 \ldots n$, that are all 0’s except the $i^{th}$ position being equal to 1. The running time of this algorithm is $O(n)$. Students who came up with this solution got half of the total score.
Problem 5. Optimal coins [15 points]

You are given a sequence of $n$ coins: $C_1, ..., C_n$.

Each coin is biased: you are told that the $i$-th coin $C_i$ has probability $p_i$ of coming up heads (call this "1") and probability $q_i = 1 - p_i$ of coming up tails (call this "0"). You are given the values of $p_i$ (and of $q_i$) for $i = 1, 2, ..., n$.

Next you are given a sequence $G_j, j = 1, 2, ..., k$ where each $G_j$ is either 1 or 0. Here $1 \leq k \leq n$.

You are asked to efficiently find an increasing subsequence $i_1, i_2, ..., i_k$ of $1, 2, ..., n$ such that when all coins are flipped independently

$$\text{Prob}(C_{i_j} = G_j \text{ for all } j = 1, 2, ..., k)$$

is maximized.

Give an efficient algorithm for finding such a subsequence that maximizes your chance of obtaining the given goal sequence $G_1, G_2, ..., G_k$ for the indices your algorithm produces.

Example: Consider a sequence of probabilities $p_1, \ldots, p_4$ equal to 0.7, 0.1, 0.2, 0.8, and a sequence $G_1, G_2$ of coins equal to 0, 1. In this case, the optimal solution $i_1, i_2$ can be seen to be equal to 2, 4, since the probability

$$\text{Prob}(C_2 = 0 \text{ and } C_4 = 1) = (1 - 0.1) \cdot 0.8 = 0.72$$

is maximized.

Solution: We will use dynamic programming.

For $1 \leq i \leq n$ and $1 \leq r \leq k$, let $P(i, r) = p_i \cdot G_r + q_i \cdot (1 - G_r)$. In addition, let $OPT(i, r)$ be the value of an optimal assignment to $G_1, \ldots, G_r$, using coins from the subsequence $C_1, \ldots, C_i$.

Then, we have that our base cases are:

$$OPT(i, 1) = \max_{1 \leq j \leq i} P(i, 1)$$

$$OPT(i, r) = 0, \text{ if } i < r$$

The recurrence is the following (for $2 \leq r \leq k$ and $r \leq i \leq n$):

$$OPT(i, r) = \max \{P(i, r) \cdot OPT(i - 1, r - 1), OPT(i - 1, r)\}$$

There are $nk$ subproblems, and based on the recursion it takes $O(1)$ to solve a problem based on the solutions of the subproblems. Hence, our running time is $O(nk)$.
Problem 6. Searching Noisy Sequences, generalized [20 points]

You are given a sequence of symbols \( t = t[0] \ldots t[n-1] \), and a pattern \( p = p[0] \ldots p[m-1] \), where the symbols \( p[i], t[i] \) come from the set \( \{0 \ldots L\} \). The goal is to match \( p \) and \( t \). That is, for a specified symbol scoring function \( sc(a, b) \) that returns the score of a match between symbols \( a \) and \( b \), the goal is to find the match score

\[
    score(s) = \sum_{i=0}^{m-1} sc(t[s+i], p[i])
\]

for all shifts \( s = 0 \ldots n - m \).

In pset 2 you have seen an algorithm (for a closely related problem) that works for the binary alphabet, i.e., \( L = 2 \). Our goal is to extend that algorithm to make it work for larger alphabet sizes.

We consider a scoring function \( sc \) such that \( sc(a, b) = 0 \) if \( a = b \) and \( sc(a, b) = 1 \) otherwise. That is, \( score(s) \) computes the number of mismatches between \( p \) (shifted by \( s \)) and \( t \).

(a) Give a deterministic algorithm that computes \( score(s) \) for all shifts \( s \). For full credit your algorithm should run in time \( O(Ln \log n) \).

Solution: Recall from pset 2 the algorithm using FFT that solves the binary pattern matching problem in \( O(n \log n) \) time. Our strategy is to find the number of matches between \( p \) and \( t \) first. This allows us to find the number of mismatches trivially by subtracting from \( m \), the length of the pattern. We will find the total number of matches by summing the number of matches of all symbols \( a \in \{0 \ldots L\} \). To do so, we replace all the occurrences of symbol \( a \) in \( p \) and \( t \) by 1, and set everything else to 0. Running the algorithm from pset 2 gives us the number of matches of symbol \( a \) for each offset. We repeat this \( L \) times for every symbol, and sum up the results. This solves the problem in \( O(Ln \log n) \) as required.

Variations:

- The above solution essentially encodes \( a \) into \( L \) bits using unary representation, i.e., the \( a \)-th bits of the encoding is set to 1 and others to 0. One could instead try binary encoding where each symbol \( a \) is mapped to its binary representation of length \( \lceil \log(L + 1) \rceil \). Unfortunately, this does not preserve the number of mismatches exactly. E.g., for \( L=15 \), the symbol 0 is mapped to 0000 while 15 is mapped to 1111, so the number of mismatched gets multiplied by 4.
(b) Give a randomized algorithm that for each $s$ provides an estimate $E(s)$ such that the expected value of $E(s)$ is equal to $\text{score}(s)/2$. Your algorithm should run in time $O(n \log n)$.

**Hint:** Consider hashing the alphabet into a smaller range, and using the algorithm from part (a).

**Solution:** We choose a uniformly random hash function $h : \{0 \ldots L\} \to \{0, 1\}$, replace each symbol $a$ by $h(a)$ in both $p$ and $t$ (obtaining sequences $h(p)$ and $h(t)$) and run the same algorithm as in pset 2. We can find the mismatches for $h(p)$ and $h(t)$ in $O(n \log n)$ time. The estimator $E(s)$ is defined as the number of mismatches between $h(p)$ (shifted by $s$) and $h(t)$.

**Analysis:** if two symbols are the same in $p$ and $t$, their hashed values must also be the same. On the other hand, if two symbols are not matched, the probability that they will be hashed to the same symbol is $1/2$. Thus, the expected number of mismatches will be reduced by $1/2$, i.e., $E(s) = \text{score}(s)/2$.

**Variations:**

- Some people used deterministic hash functions, e.g., $h(a) = a \mod 2$. Unfortunately, this works only for random input, not in the worst case. E.g., for $L = 2$, if we have $p = [0, 0, 0, 0]$ and $t = [2, 2, 2, 2, 2]$, then after hashing by $h$ all mismatches are lost.

- Another solution is to map random $L/2$ symbols to 0 and the rest of the symbols to 1. However, this solution encounters problems if $L$ is odd, since the numbers of symbols mapped to 0 and 1 are imbalanced.