Final Exam

- Do not open this exam booklet until you are directed to do so. Read all the instructions first.
- The quiz contains 6 multi-part problems. You have 180 minutes to earn 120 points.
- This quiz booklet contains 11 double-sided pages, including this one and a double-sided sheet of scratch paper; there should be 18 (numbered) pages of problems.
- This quiz is closed book. You may use three 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 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>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>120</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name: ________________________________

Circle your recitation:

- R01  R02  R03  R04  R05  R06
- F10  F11  F12  F1   F2   F3
- Joe  Joe  Khanh Khanh Emily Emily
- R07  R08  R09  R10
- F11  F12  F1   F2
- Matt Matt Geoff Geoff
Problem 1. True or False, and Justify [42 points] (14 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. If you need to make a reasonable assumption in order to answer a question (for example, if you need to assume that P ≠ NP), please state that assumption explicitly.

(a) T F [3 points] If problem A can be reduced to 3SAT via a deterministic polynomial-time reduction, and A ∈ NP, then A is NP-complete.

Solution: False. We need to reduce in the other direction (reduce an NP-hard problem to A).

(b) T F [3 points] Let G = (V, E) be a flow network, i.e., a weighted directed graph with a distinguished source vertex s, a sink vertex t, and non-negative capacity c(u, v) for every edge (u, v) in E. Suppose you find an s-t cut C which has edges e_1, e_2, ..., e_k and a capacity f. Suppose the value of the maximum s-t flow in G is f.

Now let H be the flow network obtained by adding 1 to the capacity of each edge in C. Then the value of the maximum s-t flow in H is f + k.

Solution: False. There could be multiple min-cuts. Consider the graph s-v-t where the edges have capacity 1; either edge in itself is a min-cut, but adding capacity to that edge alone does not increase the max flow.
(c) T F [3 points] Let $A$ and $B$ be optimization problems where it is known that $A$ reduces to $B$ in polynomial time. Additionally, it is known that there exists a polynomial-time 2-approximation for $B$. Then there must exist a polynomial-time 2-approximation for $A$.

**Solution:** False; approximation factor is not (necessarily) carried over in polynomial-time reduction. See e.g. set cover vs. vertex cover.

(d) T F [3 points] There exists a polynomial-time 2-approximation algorithm for the Traveling Salesman Problem.

**Solution:** False, assuming $P \neq NP$. There is an approximation algorithm in the special case where the graph obeys the triangle inequality, but we don’t know of one in general.
(e) T F  [3 points] A dynamic programming algorithm that solves $\Theta(n^2)$ subproblems could run in $\omega(n^2)$ time.

Solution: True. It could take $\omega(1)$ time per subproblem.

(f) T F  [3 points] If $A$ is a Monte Carlo program computing a predicate $f(x)$, and $B$ is a Las Vegas program computing a predicate $g(x)$, then

```python
if A(x) then
    return B(x)
else
    return False
```

is a Monte Carlo program computing $(f(x) \land g(x))$.

Solution: False. $B$ has a chance of taking arbitrarily long, so the algorithm is not Monte Carlo.
(g) T F [3 points] Dynamic programming programs require space at least proportional to the number of subproblems generated (in order to “memoize” the solution to each subproblem).

Solution: False. Under some circumstances, we can reuse the same space for multiple subproblems; for example, if each subproblem of size $k$ only looks at subproblems of size $k-1$, then when calculating bottom-up we need not store subproblems of size $k-2$ once all subproblems of size $k-1$ have been calculated. (See the discussion of longest common subsequence in CLRS.)

(h) T F [3 points] Let $H = \{h_i : \{1, 2, 3\} \rightarrow \{0, 1\}\}$ be a hash family defined as follows.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$h_2$</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$h_3$</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(For example, $h_1(3) = 0$.)

Then $H$ is a universal hash family.

Solution: False. Consider elements 1 and 3: $h_1$ and $h_2$ both cause a collision between them, so in particular a uniformly random hash function chosen from $H$ causes a collision between 1 and 3 with probability $2/3$, greater than the $1/2$ allowed for universal hashing (since there are 2 hash buckets).
(i) T F [3 points] If we use a max-queue instead of a min-queue in Kruskal’s MST algorithm, it will return the spanning tree of maximum total cost (instead of returning the spanning tree of minimum total cost). (Assume the input is a weighted connected undirected graph.)

Solution: True. The proof is essentially the same as for the usual Kruskal’s algorithm. Alternatively, this is equivalent to negating all the edge weights and running Kruskal’s algorithm.

(j) T F [3 points] Define a graph as being tripartite if its vertices can be partitioned into three sets $X_1, X_2, X_3$ such that no edge in the graph has both vertices in the same set. (That is, all edges are between vertices in different sets.) Then deciding whether a graph is tripartite can be done in polynomial time.

Solution: False, assuming $P \neq NP$. This is exactly 3-colorability; partitioning the vertices into three sets with no internal edges is the same as coloring them with three colors such that no edge has two endpoints of the same color. As seen in lecture, 3-coloring is NP-complete.
(k) T F [3 points] A randomized algorithm for a decision problem with one-sided-error and correctness probability $1/3$ (that is, if the answer is YES, it will always output YES, while if the answer is NO, it will output NO with probability $1/3$) can always be amplified to a correctness probability of 99%.

**Solution:** True. Since the error is one-sided, it in fact suffices for the correctness probability to be any constant $> 0$. We can then repeat it, say, $k$ times, and output NO if we ever see a NO, and YES otherwise. Then, if the correct answer is YES, all $k$ repetitions of our algorithm will output YES, so our final answer is also YES, and if the correct answer is NO, each of our $k$ repetitions has a $1/3$ chance of returning NO, in which case our final answer is, correctly, NO, with probability $1 - (2/3)^k$, so $k = \log_{3/2} 100$ repetitions suffice.

(l) T F [3 points] Let $B_0, B_1, B_2, \ldots$ be an infinite sequence of decision problems, where $B_0$ is known to be NP-hard and

$$B_i \leq_p B_{i+1} \text{ for all } i \geq 0.$$ 

Then it must be the case that $B_i$ is NP-hard for all $i \geq 0$.

**Solution:** True. This can be seen by induction; if $B_i$ is NP-hard, and there is a polynomial-time reduction from $B_i$ to $B_{i+1}$, then $B_{i+1}$ is NP-hard.
(m) T F [3 points] Let \( L \) be a decision problem. If there exists an interactive proof for \( L \) where the verifier is deterministic, then \( L \in NP \).

Solution: True. If the verifier is deterministic, the transcript of the interactions between the prover and verifier will always be the same. Thus, the transcript itself is a polynomial-sized certificate for \( L \).

(n) T F [3 points] Let \( L \) be a decision problem. If there exists an interactive proof for \( L \) where the prover runs in polynomial time, then \( L \in P \).

Solution: False. Either the prover or the verifier could be randomized, which would allow them to prove a larger class of problems (assuming \( P \neq BPP \)).
Problem 2. Short Answer [41 points] (9 parts)

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

(a) [7 points] Suppose that \( n \) women check their coats at a concert. However, at the end of the night, the attendant has lost the claim checks and doesn’t know which coat belongs to whom. All of the women came dressed in black coats that were nearly identical, but of different sizes. The attendant can have a woman try a coat, and find out whether the coat fits (meaning it belongs to that woman), or the coat is too big, or the coat is too small. However, the attendant cannot compare the sizes of two coats directly, or compare the sizes of two women directly. Describe how the attendant can determine which coat belongs each woman in expected \( O(n \log n) \) time. Give a brief analysis of the running time of your algorithm.

Solution: This can be solved with a modification of randomized Quicksort.

- Pick a random coat (the “pivot coat”), and have all the women try it on. Now we have the woman who fits the coat (the “pivot woman”), and a partition of the rest of the women into those who are smaller than the pivot and those who are larger.
- Have the woman who fit the coat try on all the coats. Now we have a partition of the rest of the coats into those that are smaller and those that are larger than the pivot.
- Recursively solve the two smaller subproblems – matching the women and coats that are smaller than the matched pair, and matching the women and coats that are larger than the matched pair.

This algorithm makes twice as many comparisons as randomized Quicksort does on an array of size \( n \). Therefore, the asymptotic expected running time is the same as that of randomized Quicksort – \( O(n \log n) \).
(b) [4 points] Let \( F_1, F_2, \ldots = 1, 1, 2, 3, 5, 8, \ldots \) denote the usual sequence of Fibonacci numbers (defined by \( F_1 = 1, F_2 = 1 \), and \( F_i = F_{i-1} + F_{i-2} \) for \( i > 2 \)).

Suppose that a file to be compressed contains \( k \) different symbols \( a_1, a_2, \ldots, a_k \) and that it contains \( F_i \) occurrences of \( a_i \) for each \( i \). Thus, if \( k = 4 \), the string has length 7 and contains 2 occurrences of \( a_3 \).

Assume the file is encoded with Huffman encoding. How many bits will be used to encode \( a_i \), as a function of \( i \) and/or \( k \)? State your answer concisely. You do not need to provide a proof.

**Solution:** \((k + 1) - i \) for \( i > 1 \); \( k - 1 \) for \( i = 1 \)

---

(c) [4 points] As a final project for one of your other Course 6 classes, you have a massive program to run. After much effort, you are able to parallelize 90\% of your code. The computer lab has two systems on which you could run your program:

- a cluster of 90 single-core computers each running at 1GHz, and
- a computer with 9 cores each running at 2GHz.

Which one should you choose to complete your project as quickly as possible?

**Solution:** Compared to a single 1GHz single-core machine, the first option offers a speedup factor of

\[
\frac{1}{1 + .9/90} = \frac{1}{.11} \approx 9,
\]

while the second offers a speedup factor of

\[
\frac{2}{1 + .9/9} = \frac{2}{.2} = 10.
\]

So you should go with the second.
(d) [5 points] Recall the clique problem from lecture: Given an undirected graph \( G = (V, E) \) and a positive integer \( k \), is there a subset \( C \) of \( V \) of size at least \( k \) such that every pair of vertices in \( C \) has an edge between them?

Ben Bitdiddle thinks he can solve the clique problem in polynomial time using linear programming.

- Let each variable in the linear program represent whether or not each vertex is a part of our clique. Add constraints stating that each of these variables must be nonnegative and at most one.
- We go through the graph \( G \) and consider each pair of vertices. For every pair of vertices where there is not an edge in \( G \), add a constraint stating that the sum of the variables corresponding to the endpoint vertices must be at most one. This ensures that both of them cannot be part of a clique if there is no edge between them.
- The objective function is the sum of the variables corresponding to the vertices. We wish to maximize this function.

Ben argues that the value of the optimum must be the size of the maximum-size clique in \( G \), and we can then simply compare this value to \( k \). Explain the flaw in Ben’s logic.

**Solution:** This is an integer program, not a linear program, and therefore we don’t know how to solve it in polynomial time. (Alternatively, if we don’t add the integrality constraints, we can solve it in polynomial time but will likely get fractional values back; it’s unclear what fractional values of the variables mean with regard to the clique.)
(e) [4 points] In a weighted connected undirected graph that might have negative-weight edges but no negative-weight cycles, how would you find a triple of distinct vertices \( x, y, z \) that minimizes \( f(x, y, z) = d(x, y) + d(y, z) + d(z, x) \) where \( d(u, v) \) is the length of the shortest path from \( u \) to \( v \)?

The running time of your algorithm should be \( O(n^3) \), where \( n \) is the number of vertices in the graph.

**Solution:** Run Johnson’s all-pairs shortest-paths algorithm to find all of the shortest paths \( d(u, v) \). This takes \( O(V^2 \log V + VE) = O(n^3) \) time, since \( E = O(n^2) \). Then calculate \( f \) for all triples of vertices in the graph, and take the minimum. There are \( O(n^3) \) triples, and \( f \) can be calculated in \( O(1) \) time given the \( d(u, v) \) values, so this step also takes \( O(n^3) \) time.

(f) [4 points] Suppose you are using RSA and you change your public key \((e, N)\) every so often, where \( N = pq \) is the product of your two large secret primes.

Why is it not a good idea to leave \( p \) the same and just replace \( q \) with a different secret prime \( q' \) (so your new \( N' \) is just \( pq' \))?  

**Solution:** Anyone could compute the GCD of two of your public keys (using the Euclidean algorithm, which is polynomial-time) to find \( p \), and thus factor \( N \) and \( N' \).
(g) [7 points] You are working at a hospital trying to diagnose patients; you may assume that each patient has exactly one disease. You know of $m$ different diseases $d_1, d_2, \ldots, d_m$. You have $n$ different tests you can run (labeled $T_1, T_2, \ldots, T_n$), each of which comes up positive for some set of diseases and negative for other diseases. You would like to correctly diagnose all patients while giving them the minimum necessary number of tests—or, at least, close to the minimum number. Since you must send the tests to the lab for processing, all tests must be performed in parallel.

We say that a set of tests $T \subseteq \{T_1, T_2, \ldots, T_n\}$ is comprehensive if, for every pair of diseases $(d_i, d_j)$, there is some test $T_k \in T$ that distinguishes them—that is, it returns positive for one and negative for the other. The minimum-comprehensive-set problem (MCS) is the problem of finding a comprehensive set of tests of minimum cardinality. MCS is known to be NP-hard.

Describe a polynomial-time $\alpha$-approximation algorithm for the MCS problem, where $\alpha = \ln(m(m - 1)/2)$.

**Solution:** For every pair of diseases, there is at least one of the tests that distinguishes them, and we want a minimum-cardinality set of the tests that between them distinguish all diseases. This is simply the Set Cover problem operating on pairs of diseases; we can use the standard approximation for Set Cover seen in CLRS/lecture.

(h) [3 points] State the three properties a trapdoor function should have.

**Solution:** Easy to compute, hard to invert without the trapdoor information, easy to invert with the trapdoor information.
(i) [4 points] Suppose you are given a polynomial time algorithm DECISION-FACTOR that, given two integers \( k \) and \( n \), returns \textsc{YES} if \( n \) has a prime factor less than \( k \), and \textsc{NO} if \( n \) does not. Give a polynomial time algorithm for computing a single prime factor of \( n \).

**Solution:** Use DECISION-FACTOR in a binary search to find the smallest prime factor \( p \) of \( n \): for every \( m \leq p \) we have DECISION-FACTOR(\( m \)) = \textsc{NO} and for every \( m > p \) we have DECISION-FACTOR(\( m \)) = \textsc{YES}. This takes \( O(\log n) \) calls to DECISION-FACTOR, which is polynomial (linear, in fact) in the length of the input \( n \), so the overall running time is polynomial as well.
Problem 3. More Spy Games [9 points]

An enemy country, Elbonia, has $n$ transmitter/receiver pairs $(t_i, r_i)$. You can model the position of each $t_i$ and each $r_i$ as a point in the plane. Enemy communications travel along the straight-line segment from $t_i$ to $r_i$. You can place eavesdrop units at any point in the plane, but a unit must be on the line segment from $t_i$ to $r_i$ in order to eavesdrop successfully. If you put a unit at the intersection of two such segments, that unit can eavesdrop on both transmitter/receiver pairs. Assume no three such segments intersect at a point.

Your intelligence agency has given you a list of the coordinates of all $n$ enemy transmitter/receiver pairs. Briefly describe a polynomial time algorithm for finding the minimum number of eavesdrop units required to eavesdrop on all $n$ transmitter/receiver pairs. (No proof needed.)

Solution: Construct a graph $G$ consisting of a vertex $v_i$ for each transmitter/receiver pair $(t_i, r_i)$, and an edge between vertices $v_i$ and $v_j$ if the corresponding line segments intersect. This can be done in $O(n^2)$ time. Now, any vertex that is isolated must be eavesdropped on by its own dedicated unit, and we can remove it from consideration. The problem then reduces to finding a minimum edge cover of $G$, that is, the minimum number of edges such that every vertex in $G$ is incident on at least one.

We can do this by first finding a maximum matching in $G$ (using any of several matching algorithms covered in class, all of which run in polynomial time), and adding an edge to cover each of the remaining uncovered vertices. To see why this indeed achieves a minimum edge cover, observe that any edge cover contains a matching, each edge of which covers two vertices, together with some additional edges, each of which covers a single additional vertex. Thus the smallest edge cover we can hope to obtain comprises, in this manner, of a maximum matching of $G$ together with an edge for each remaining unmatched vertex. But this is indeed what we construct, so it must be the minimum edge cover, and we are done.
Problem 4. Almost Sorted [9 points]

A sequence $x_1, x_2, \ldots, x_n$ of real numbers is said to be **sorted** if

$$x_1 \leq x_2 \leq \cdots \leq x_n.$$  

We say that $x_1, x_2, \ldots, x_n$ is **D-almost-sorted** for a non-negative real number $D$ if there exists another sequence $y_1, y_2, \ldots, y_n$ of real numbers such that $y_1, y_2, \ldots, y_n$ is sorted, and $\sum_i |x_i - y_i| \leq D$. (That is, by “shifting” values $x_i$ to new values $y_i$, such that the total amount of shifting is at most $D$, the new set of numbers is sorted.)

Describe concisely a polynomial-time algorithm which, given an input sequence $x_1, x_2, \ldots, x_n$ and a non-negative real number $D$, determines whether $x$ is $D$-almost-sorted.

**Solution:** We solve this problem using linear programming. To determine whether a sequence $x_1, x_2, \ldots, x_n$ is $D$-almost-sorted, check whether the following LP is feasible:

\[
\begin{align*}
\text{minimize} & \quad 0 \\
\text{subject to} & \quad x_i = y_i + c_i - d_i \quad \text{for } i = 1, \ldots, n \\
& \quad y_i \leq y_{i+1} \quad \text{for } i = 1, \ldots, n - 1 \\
& \quad \sum_i (c_i + d_i) \leq D
\end{align*}
\]

(The objective function is irrelevant.) Alternatively, we could solve the following LP, and then check whether the optimal value of the objective function is at most $D$.

\[
\begin{align*}
\text{minimize} & \quad \sum_i (c_i + d_i) \\
\text{subject to} & \quad x_i = y_i + c_i - d_i \quad \text{for } i = 1, \ldots, n \\
& \quad y_i \leq y_{i+1} \quad \text{for } i = 1, \ldots, n - 1
\end{align*}
\]

Linear programming can be solved in worst-case polynomial time by the ellipsoid algorithm or interior-point methods.
Problem 5. Randomized 3-Coloring [8 points] (3 parts)

In an undirected graph $G = (V, E)$, a coloring is a mapping $c$ which assigns colors to vertices. We denote the color of vertex $v$ by $c(v)$.

We say a coloring $c$ satisfies an edge $e = (u, v)$ if $c(u) \neq c(v)$ (that is, the endpoints of the edges are assigned different colors). Let the function $s(c)$ count the number of satisfied edges under a coloring $c$.

Define the 3-coloring optimization problem as follows: Given an undirected graph $G = (V, E)$, output a coloring $c$ such that $c(v) \in \{R, W, B\}$ for all $v \in V$, such that $s(c)$ is maximized.

Here is one very simple randomized algorithm:

**\textsc{Randomized-Color}(G)**

1. for each $v \in V$
2. Pick a color uniformly at random in $\{R, W, B\}$
3. Let $c(v) =$ color picked
4. return $c$

(a) [2 points] Let $e$ be any edge. What is the probability that the coloring picked satisfies $e$?

\textbf{Solution:} $\frac{2}{3}$
(b) [2 points] What is the expected number of edges satisfied by the coloring produced by $c$? Justify.

Solution: $2|E|/3$, due to linearity of expectation over all edges.

(c) [4 points] Show that RANDOMIZED-COLOR is a polynomial-time randomized $(3/2)$-approximation algorithm for the 3-coloring optimization problem. That is, show that $E(s(c)) \geq (2/3)s(c^*)$ where $c^*$ is the optimal coloring.

Solution: The optimal coloring can satisfy at most $|E|$ edges, so $s(c^*) \leq |E|$. From (b), $E[s(c)] = 2|E|/3$. Thus, $E(s(c)) \geq (2/3)s(c^*)$. 
Problem 6. Sublinear-Time Unimodal Testing [10 points]

We say that an array \( A[1..n] \) of real numbers is **unimodal** if there exists an integer \( k \) such that \( 1 \leq k \leq n \), \( A[1..k] \) is monotonically non-decreasing, and \( A[k..n] \) is monotonically non-increasing.

We say that \( A \) is \( \epsilon \)-far from being unimodal if you have to remove more than \( \epsilon n \) elements from \( A \) in order for the remaining sequence to be unimodal.

Give a sublinear-time property tester that, given an array \( A[1..n] \) of distinct real numbers:

- if \( A \) is unimodal, outputs **YES** with probability 1, and
- if \( A \) is \( \epsilon \)-far from being unimodal, outputs **NO** with probability at least \( 2/3 \).

**Solution:** We first use binary search to find a candidate \( k \): each query will be of two consecutive indices, to see if we are to the left of \( k \) or to the right. Since \( A \) consists of all distinct values, we will never get a tie. Then, once we have such a \( k \), we run our monotonicity tester with parameter \( \epsilon \) on \( A[1..k] \) and \( A[k..n] \). If both return **YES**, we return **YES**, otherwise we return **NO**.

To see that this is correct, suppose that \( A \) is unimodal. Then our search for \( k \) must return the correct \( k \), and \( A[1..k] \) and \( A[k..n] \) must both be monotone, so our subroutines both return **YES** and we return **YES** as well, as required. On the other hand, if \( A \) is \( \epsilon \)-far from being unimodal, meaning we need to remove \( \epsilon n \) elements from \( A \) to make it unimodal, then no matter what \( k \) we pick, either \( A[1..k] \) or \( A[k..n] \) must be \( \epsilon \)-far from being monotone. Specifically, suppose, to get a contradiction, that \( A[1..k] \) and \( A[k..n] \) are both at most \( \epsilon' \)-far from being monotone, for some \( \epsilon' < \epsilon \). This means we can remove \( \epsilon' k \) elements from \( A[1..k] \) to make it monotone, and \( \epsilon'(n-k) \) elements from \( A[k..n] \) to make it monotone. But then we could remove these same \( \epsilon' k + \epsilon'(n-k) = \epsilon' n < \epsilon n \) elements from \( A[1..n] \) to make it unimodal, contradicting the fact that \( A \) is \( \epsilon \)-far from being unimodal. It follows that at least one of our subroutines must return **NO** with probability at least \( 2/3 \), so we also return **NO** with probability at least \( 2/3 \).

Note that our monotonicity tester allows for a specification of the direction of monotonicity (i.e., increasing or decreasing).