Final Exam Solutions

<table>
<thead>
<tr>
<th>Problem</th>
<th>Parts</th>
<th>Points</th>
<th>Grade</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>180</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 1. True or False, and Justify [80 points] (16 parts)

Circle T or F for each of the following statements to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

T F For every two positive functions $f$ and $g$, if $g(n) = O(n)$, then $f(g(n)) = O(f(n))$.

Solution: False.

Let $f(n) = 2^n$ and $g(n) = 2n = O(n)$. Then $f(g(n)) = 2^{2n} = (2^n)^2 = (f(n))^2 \neq O(f(n))$.

T F Suppose $f(n) = 4f(n/4) + n$ for $n > 8$, and $f(n) = O(1)$ for $n \leq 8$. Similarly, suppose $g(n) = 3g(n/4) + n\lg n$ for $n > 8$, and $g(n) = O(1)$ for $n \leq 8$. Then $f(n) = \Theta(g(n))$.

Solution: True.

By the Master Method (Cases 1 and 3), both are $\Theta(n \lg n)$. 
Suppose that a randomized algorithm $A$ has expected running time $\Theta(n^2)$ on any input of size $n$. Then it is possible for some execution of $A$ to take $\Omega(2^n)$ time.

**Solution:** True.

Consider a (somewhat dumb) sorting algorithm that first sorts $n$ items using mergesort, in time $\Theta(n \lg n)$, and then flips $n$ (fair) coins. If at least one coin turns up heads, the algorithm waits $n^2$ additional units of time before terminating. If all coins turn up tails, it waits $2^n$ units of time. Although in the latter case the running time is $\Omega(2^n)$, the expected running time is only $\Theta(n \lg n) + (1 - 2^{-n})(n^2) + 2^{-n}(2^n) = \Theta(n^2)$.

Suppose we maintain a hash table with $m$ slots using chaining and a hash function chosen from a universal hash family. If we insert $n > m$ keys into this (initially empty) hash table, then the total number of collisions is $O(n/m)$ in expectation. (Recall that a collision is a pair of distinct keys that hash to the same slot.)

**Solution:** False.

In expectation, any particular key $x$ collides with $\Theta(n/m)$ other keys. Thus, summing over all $n$ keys, we have a total of $\Theta(n^2/m) \neq O(n/m)$ collisions.
**T F** Building an $n$-element heap requires $\Theta(n \lg n)$ time.

**Solution:** False.

It takes $\Theta(n)$ time. See CLRS, pages 133–135.

**T F** Given an unsorted array $A$ of $n$ integers, let $x_i$ denote the $2^i$th smallest element in $A$. Then we can compute $\sum_{i=0}^{\lfloor \lg n \rfloor} x_i$ in $O(n)$ time.

**Solution:** True.

Initialize $i \leftarrow \lfloor \lg n \rfloor$, $\tilde{A} \leftarrow A$, and $s \leftarrow 0$, and then repeat the following until $i < 0$: compute the $2^i$th element $x_i$ of $\tilde{A}$ using SELECT, set $s \leftarrow s + x_i$, update $\tilde{A}$ to be those elements of $\tilde{A}$ that are smaller than $x_i$, and update $i$ to be $i - 1$. At the end, return $s$. Each iteration takes time linear in the size of $\tilde{A}$. In the first iteration (in which $i = \lfloor \lg n \rfloor$), $\tilde{A}$ has $n$ elements, so this iteration takes $cn$ time (for some appropriate constant $c > 0$); in each subsequent iteration, $\tilde{A}$ has $2^{i+1} - 1$ elements, so the time taken is $c(2^{i+1} - 1)$. Thus the total time taken is

$$cn + \sum_{i=0}^{\lfloor \lg n \rfloor - 1} c(2^{i+1} - 1) = \Theta(n) .$$
T  F  Suppose that you have a 2-3 tree $T_1$ and AA-tree (from problem set 4) $T_2$, each storing the same set of keys. Then in-order traversals of $T_1$ and $T_2$ can result in different sequences of keys.

Solution: False.
Both 2-3 trees and AA trees are search trees. Thus, an in-order traversal of either produces the (same) sorted sequence of keys.

T  F  There are at least two distinct 2-3 trees containing keys 1, 2, 3, 4, 5.

Solution: True. E.g., root node could have either 2 or 3 children.
T F Graduating from MIT requires passing $n$ specified classes. You decide to take each class every semester until you pass it. Suppose that, every semester you take a class, you have a 50% chance of passing it and a 50% chance of having to drop it. Then you will graduate in $O(\lg n)$ semesters with high probability.

**Solution:**  True.

For each of the $n$ courses, the probability that do not pass the course after $k \lg n$ semesters is $(1/2)^{k \lg n} = 1/n^k$. Therefore, the probability that there exists a course that you do not pass after $k \lg n$ semesters is at most $1/n^{k-1}$. Thus you graduate with probability at least $1 - 1/n^{k-1}$. Choosing $k$ (and thus $k - 1$) to be an arbitrarily large constant, the number of semesters is $O(\lg n)$ with high probability. (This derivation is essentially the same as Problem 2 on Problem Set 5.)
Suppose that you have two deterministic online algorithms, $A_1$ and $A_2$, with competitive ratios $c_1$ and $c_2$, respectively. Consider the randomized online algorithm $A^*$ that flips a fair coin once at the beginning; if the coin comes up heads, it runs $A_1$ from then on; if the coin comes up tails, it runs $A_2$ from then on. Then the expected competitive ratio of $A^*$ is at least $\min\{c_1, c_2\}$.

**Solution:** False.

Randomization can help a lot. In particular, the expected competitive ratio of $A^*$ can be smaller than $\min\{c_1, c_2\}$. In class we have seen several examples of problems where this happens. For instance, see Problem 2 on Problem Set 6.
T F Reweighting a graph with negative edge weights but no negative-weight cycles, as in John-
son’s algorithm, can be used to solve the single-source shortest-paths problem more effi-
ciently than Bellman-Ford.

Solution: False.
Johnson’s reweighting uses the Bellman-Ford algorithm as a subroutine, to solve a sys-
tem of difference constraints. Therefore Johnson’s reweighting cannot more efficient than
Bellman-Ford.

T F Every problem in NP can be solved in exponential time.

Solution: True.
The brute-force search algorithm can solve every such problem in exponential time. This
algorithm guesses a certificate $y$ of polynomial length $n^c$, and runs the verification algo-
rithm on the input $x$ and certificate $y$ in $n^{c'}$ time. If the verification algorithm outputs 1
for any certificate, the brute-force algorithm returns 1; otherwise, it returns 0. The running
time of this algorithm is $O(2^{n^c} n^{c'})$, which is $O(2^{n^{c''}})$ for a suitable constant $c''$. 
T F For any decision problem $\pi$ in NP, define the input size $n$ as the parameter $k$ to fix. Then $\pi$ is fixed-parameter tractable with respect to $n$.

**Solution:** True.
Recall that a parameterized problem is fixed-parameter tractable with respect to a parameter $k$ if the problem can be solved in $f(k) \cdot n^c$ time for some constant $c$. Clearly, problems in NP are FPT with respect to $n$, because by the previous part they can be solved in $f(n) = O(2^n)$ time for some constant $c$.

T F Define an independent set of a graph $G = (V, E)$ to be a subset $S \subseteq V$ of vertices such that $V - S$ is a vertex cover of $G$. Every 2-approximation algorithm for finding a minimum vertex cover is also a 2-approximation algorithm for finding a maximum independent set.

**Solution:** False.
Let $G$ be a cycle on six vertices. Clearly, both the maximum independent set and the minimum vertex cover of $G$ are of size 3. A 2-approximation to minimum vertex cover may have size up to 6, but its complement will not necessarily be a 2-approximation to maximum independent set, having size as low as 0.
Problem 2. One, One Room; Two, Two Rooms; Ah Ha Ha! [20 points] (3 parts)

You are maintaining a hotel room reservation system for an $n$-day period, with dates labeled 1, 2, ..., $n$. Your reservation system must support two operations:

- **RESERVE($i, j$)** makes a room reservation for the dates $i, i + 1, \ldots, j$.
- **COUNT($i$)** computes how many rooms are currently reserved on day $i$.

Your goal is to construct a data structure that supports both operations in $O(\log n)$ time. Assume that $n$ is an exact power of two.

You decide to maintain your data in the form of a perfectly balanced binary tree. The root corresponds to the entire interval $[1 \ldots n]$; the root’s left child corresponds to the interval $[1 \ldots n/2]$; the root’s right child corresponds to the interval $[(n/2 + 1) \ldots n]$; etc. At the (bottom) leaf level, each leaf corresponds to a single day, and the leaf order matches the day order. Thus there are exactly $n$ leaves and $1 + \log n$ levels.

(a) What additional information would you maintain in the nodes in order to support the updates and queries efficiently?

**Solution:** For each node $x$ representing some interval $I_x$ of dates, we store a non-negative integer $c[x]$. This value $c[x]$ counts the number of reservation intervals $[i, j]$ that wholly contain $I_x$ but do not wholly contain the parent interval $I_{p[x]}$. 
(b) Briefly describe how you would implement the \textsc{Count}(i) operation. Briefly justify why the running time is $O(\lg n)$.

\textbf{Solution:} \textsc{Count}(i) walks the path $x_0, x_1, x_2, \ldots, x_{\lg n}$ from the leaf $x_0$ corresponding to date $i$ up to the root $x_{\lg n}$ of the tree. It returns the sum of the $c$ values along this path: $c[x_0] + c[x_1] + c[x_2] + \cdots + c[x_{\lg n}]$. This computation takes $O(\lg n)$ time because we spend $O(1)$ time at each node along the path of length $1 + \lg n$.

We claim that this sum equals the number of reservation intervals containing date $i$. To see this, consider a reservation interval $I$ containing $i$. The intervals $I_0, I_1, I_2, \ldots, I_{\lg n}$ along the leaf-to-root path are nested: $I_0 \subseteq I_1 \subseteq I_2 \subseteq \cdots \subseteq I_{\lg n}$. Hence, there is a unique $k$ such that $I$ wholly contains $I_k$ but $I$ does not wholly contain the parent interval $I_{k+1}$. Thus, $c[x_k]$ counts $I$, and no other $c[x_k']$ counts $I$.

(c) Briefly describe how you would implement the \textsc{Reserve}(i, j) operation. Briefly justify why the running time is $O(\lg n)$.

\textbf{Solution:} \textsc{Reserve}(i, j) simultaneously walks up the tree from the two leaves $x_0$ and $y_0$ corresponding to $i$ and $j$, visiting nodes $x_0, x_1, x_2, \ldots$ and $y_0, y_1, y_2, \ldots$, until the two paths meet at some node $x_k = y_k$, the lowest common ancestor of $x_0$ and $y_0$. We define a \texttt{test} operation on a node $x$: if $[i, j]$ wholly contains $x$’s interval but not $x$’s parent’s interval, then we increment $c[x]$. Then \textsc{Reserve}(i, j) tests $x_k = y_k$, and tests each child of every node $x_0, y_0, x_1, y_1, \ldots, x_{k-1}, y_{k-1}, x_k = y_k$. It is easy to see that these $O(\lg n)$ nodes are all that we need to test, and that the running time is thus $O(\lg n)$.
Problem 3. Forty Two [30 points] (4 parts)

Professor Hackermann has finally cracked the meaning of life, the universe, and everything: it is $H(42, 42)$ where the function $H(m, n)$ is defined by the following unusual recurrence:

\[
\begin{align*}
H(m, 1) &= m^2; \\
H(1, n) &= n^3; \\
H(m, n) &= H(\text{FOO}(m, n), n) + H(m, \text{BAR}(m, n)) \quad \text{for all other values of } m, n \geq 1.
\end{align*}
\]

Professor Hackermann knows how to compute $\text{FOO}(m, n)$ and $\text{BAR}(m, n)$ in $O(1)$ time for given values of $m, n \geq 1$. The catch is that $\text{FOO}(m, n)$ and $\text{BAR}(m, n)$ can sometimes be larger than $m$ and $n$, so the recurrence does not obviously terminate. Nonetheless, both $\text{FOO}(m, n)$ and $\text{BAR}(m, n)$ have value 1 often enough that the recursive formula may allow computing $H(m, n)$.

(a) The professor hires you to compute $H(3, 4)$ by hand, using the following information about $\text{FOO}$ and $\text{BAR}$:

\[
\begin{align*}
\text{FOO}(3, 4) &= 7; \quad \text{BAR}(3, 4) = 1; \\
\text{FOO}(7, 4) &= 10; \quad \text{BAR}(7, 4) = 2; \\
\text{FOO}(10, 4) &= 1; \quad \text{BAR}(10, 4) = 1; \\
\text{FOO}(7, 2) &= 1; \quad \text{BAR}(7, 2) = 1.
\end{align*}
\]

Show your work.

Solution:

\[
\begin{align*}
H(7, 2) &= H(1, 2) + H(7, 1) = 8 + 49 = 57, \\
H(10, 4) &= H(1, 4) + H(10, 1) = 64 + 100 = 164, \\
H(7, 4) &= H(10, 4) + H(7, 2) = 164 + 57 = 221, \\
H(3, 4) &= H(7, 4) + H(3, 1) = 221 + 9 = 230.
\end{align*}
\]
To understand whether \( H(42, 42) \) can be computed with the recursive formula, Professor Hackermann sets out to understand which pairs \((m', n')\) arise from the recursion. The professor defines \( \text{descendants}(m, n) \) to be the set of pairs \((m', n')\) for which \( H(m', n') \) is required to compute \( H(m, n) \), i.e.,

\[
\text{descendants}(m, n) = \left\{ \left( \text{FOO}(m, n), n \right), \left( m, \text{BAR}(m, n) \right) \right\} \\
\cup \text{descendants}(\text{FOO}(m, n)) \cup \text{descendants}(m, \text{BAR}(m, n)).
\]

Note that \( \text{descendants}(m, n) \) does not necessarily include \((m, n)\).

(b) To thwart critics who claim that pairs \((m', n')\) in \( \text{descendants}(m, n) \) can grow without bound, Professor Hackermann makes the following conjecture:

**Conjecture 1** For every \( m, n \geq 1 \), and for every pair \((m', n') \in \text{descendants}(m, n)\), we have both \( m' \leq (m + n)^3 \) and \( n' \leq (m + n)^3 \).

Give an algorithm that, on input \( m, n \geq 1 \), determines whether Conjecture 1 is true for this pair of integers, i.e., whether \( m', n' \leq (m + n)^3 \) for every pair \((m', n') \in \text{descendants}(m, n)\). Your algorithm must run in time polynomial in \( m + n \).

**Solution:** To verify Conjecture 1 for a given pair of integers \( m \) and \( n \), we recursively compute the pairs that belong to \( \text{descendants}(m, n) \), using an \((m+n)^3 \times (m+n)^3\) table \( A \) to track the pairs \((m', n')\) we have already visited. When expanding the recursive definition of \( \text{descendants}(m, n) \), if we ever discover a pair \((m', n')\) out of range, (i.e., either \( m' > (m + n)^3 \) or \( n' > (m + n)^3 \)), we know that Conjecture 1 is false. On the other hand, if all elements we check are in range, then Conjecture 1 holds for \((m, n)\).

We fill in the table recursively, but use memoization to avoid recomputing repeated subproblems.

```plaintext
for i ← 1 to \((m + n)^3\)  ▷ Initialize table
  do for j ← 1 to \((m + n)^3\)
    do A[m, n] ← FALSE
  return CHECK-C1(m, n, m, n)

CHECK-C1(m', n', m, n)  ▷ Check Conjecture 1
  if m' > \((m + n)^3\) or \( n' > (m + n)^3 \)
    then return FALSE
  if A[m', n']
    then return TRUE
  else A[m', n'] ← TRUE  ▷ Do work only if we have not visited \((m', n')\)
    t₁ ← CHECK-C1(FOO(m', n'), n', m, n)
    t₂ ← CHECK-C1(m', BAR(m', n'), m, n)
  return (t₁ and t₂)
```
The amortized cost to fill any single element in the table $A$ is $O(1)$, because we can charge the cost of making recursive calls (Lines 7–9) and the cost of checking the base cases (Lines 1–4) to each table element of $A$ that is filled in (Line 6). Because we have $(m + n)^6$ table elements, the running time of the algorithm is $O((m + n)^6)$. 
(c) More critics claim that the professor’s recursive formula is useless because it could be cyclic: the computation of $H(m, n)$ could require the computation of $H(m, n)$ itself, leading to an infinite recursion. To thwart these critics, Professor Hackermann makes another conjecture:

**Conjecture 2**  For every $m, n \geq 1$, we have $(m, n) \notin \text{descendants}(m, n)$.

Give an algorithm that, on input $m, n \geq 1$, determines whether Conjecture 2 is true for this pair of integers, i.e., whether $(m, n) \notin \text{descendants}(m, n)$. Your algorithm must run in time polynomial in $m + n$, and it may assume that Conjecture 1 holds.

**Solution:** Assuming Conjecture 1 holds, all pairs $(m', n') \in \text{descendants}(m, n)$ are represented in the $(m+n)^3 \times (m+n)^3$ table $A$. Thus, we can use a memoized recursive algorithm similar to the one in part (b), except with the added check for whether we encounter $m' = m$ and $n' = n$ after the first step.

1. for $i \leftarrow 1$ to $(m+n)^3$ \hspace{1cm} ▷ Initialize table
2. \hspace{0.5cm} do for $j \leftarrow 1$ to $(m+n)^3$
3. \hspace{1.5cm} do $A[m, n] \leftarrow \text{FALSE}$
4. \hspace{1.5cm} $t_1 \leftarrow \text{CHECK-C2}(\text{FOO}(m, n), n, m, n)$
5. \hspace{1.5cm} $t_2 \leftarrow \text{CHECK-C2}(m, \text{BAR}(m, n), m, n)$
6. \hspace{1.5cm} return $(t_1 \text{ and } t_2)$

\[ \text{CHECK-C2}(m', n', m, n) \quad \text{▷ Check Conjecture 2} \]

1. if $(m', n') = (m, n)$
2. \hspace{0.5cm} then return \text{FALSE}
3. \hspace{0.5cm} if $A[m', n']$
4. \hspace{1cm} then return \text{TRUE}
5. \hspace{1cm} else $A[m', n'] \leftarrow \text{TRUE} \quad \text{▷ Do work only if we have not visited } (m', n')$
6. \hspace{1.5cm} $t_1 \leftarrow \text{CHECK-C2}(\text{FOO}(m', n'), n', m, n)$
7. \hspace{1.5cm} $t_2 \leftarrow \text{CHECK-C2}(m', \text{BAR}(m', n'), m, n)$
8. \hspace{1.5cm} return $(t_1 \text{ and } t_2)$

As before, the running time is $O((m + n)^6)$, proportional to the size of the table.
(d) Assuming Conjectures 1 and 2 hold, give an algorithm to compute $H(m, n)$ with running time polynomial in $m + n$. What is the asymptotic running time of your algorithm?

Solution: As in parts (b) and (c), we use a recursive memoized algorithm to compute $H(m, n)$. By Conjecture 1, we know we only need a table of size $(m + n)^3 \times (m + n)^3$ to store all $H$ values we might need to compute $H(m, n)$. By Conjecture 2, we know that none of the $H(m', n')$ are defined in terms of themselves. Thus, the following algorithm computes $H(m, n)$:

```
1 for i ← 1 to $(m + n)^3$ ▷ Initialize table
2 do for j ← 1 to $(m + n)^3$
3   do $A[m, n] ← \text{NIL}$
4 return COMPUTE-H(m, n)

   COMPUTE-H(m', n')  ▷ Compute $H(m', n')$
1   if $A[m', n'] = \text{NIL}$  ▷ Do work only if we have not computed $H(m', n')$ before.
2    then $v_1 ← \text{COMPUTE-H(FOO}(m', n'), n')$
3    $v_2 ← \text{COMPUTE-H(m', BAR}(m', n'))$
4    $A[m', n'] ← v_1 + v_2$
5  return $A[m', n']$
```

As before, the running time is $O((m + n)^6)$, proportional to the size of the table.
Problem 4. Cliquish Behavior [20 points] (4 parts)

Prof. Vernon has come up with the following divide-and-conquer algorithm, BREAKFAST, for finding a clique in an undirected graph $G = (V, E)$:

1. Number the vertices in $V$ as $1, 2, \ldots, n$, where $n = |V|$.
2. If $n = 1$, return $V$.
3. Partition the vertices into the two sets $V_1 = \{1, 2, \ldots, \lfloor n/2 \rfloor\}$ and $V_2 = \{\lceil n/2 \rceil + 1, \ldots, n\}$.
4. Let $G_1$ be the subgraph of $G$ induced by $V_1$, and similarly let $G_2$ be the subgraph of $G$ induced by $V_2$. (In other words, the edges of $G_1$ are all edges of $G$ that connect pairs of vertices in $V_1$, and the edges of $G_2$ are those of $G$ that connect pairs of vertices in $V_2$.)
5. Recursively find cliques $C_1 = \text{BREAKFAST}(G_1)$ and $C_2 = \text{BREAKFAST}(G_2)$.
6. Combine these two cliques as follows:
   - Initialize $C_1^+ \leftarrow C_1$ and $C_2^+ \leftarrow C_2$.
   - For every vertex $v \in C_2$, if $v$ is adjacent to every vertex of $C_1^+$, then add $v$ to $C_1^+$.
   - For every vertex $u \in C_1$, if $u$ is adjacent to every vertex of $C_2^+$, then add $u$ to $C_2^+$.
   - Return the larger of $C_1^+$ and $C_2^+$.

(a) Briefly argue that the BREAKFAST algorithm always returns a clique of $G$.

Solution: We induct on $n$, the number of nodes in the graph.

Base case: For $n = 1$, the algorithm returns a single vertex, which is a clique.

Induction step: Let $n > 1$, and assume the claim for all graphs with less than $n$ vertices. Let $G$ be a graph with $n$ vertices. Clearly, both $G_1$ and $G_2$ have less than $n$ vertices each. By the induction hypothesis, $C_1$ and $C_2$ are cliques of $G_1$ and $G_2$, respectively; as a consequence, they are both cliques of $G$ as well. Thus $C_1^+$ and $C_2^+$ are both initialized to cliques of $G$ in Step 6. Now, a vertex $v \in C_2$ is added to $C_1^+$ if and only if $v$ is adjacent to every vertex of $C_1^+$. Thus, $C_1^+$ remains a clique of $G$. Similarly, $C_2^+$ remains a clique of $G$. Because the algorithm returns either $C_1^+$ or $C_2^+$, it always returns a clique of $G$.

(b) Give an asymptotically tight upper bound on the running time of the BREAKFAST algorithm.

Solution: The worst-case running time $T(n)$ of the algorithm on an input graph with $n$ nodes satisfies the recurrence $T(n) = 2T(n/2) + \Theta(n^2)$ because the combination step in Step 6 takes $\Theta(n^2)$ time in the worst case. By Case 3 of the Master Theorem, $T(n) = \Theta(n^2)$.

---

1For a graph $G$, a clique $C \subset V$ is a subset of vertices that are all interconnected by edges.
(c) Give an example of a graph $G$ where the algorithm produces a clique of less than maximum size.

**Solution:** There are many such counterexamples. Our favorite (found by many students) is the following:

![Graph Diagram]

The maximum clique in this graph is $\{2, 3, 4, 5\}$, but the BREAKFAST algorithm returns either $\{1, 2, 3\}$ or $\{4, 5, 6\}$.

(d) If the professor could modify algorithm BREAKFAST so as to find the largest clique without increasing the asymptotic running time, what would this tell you about the classes P and NP? Briefly explain your answer.

**Solution:** This would imply that $P = NP$. If the BREAKFAST algorithm could be modified to correctly find a maximum clique in $O(n^2)$ time, we would have a polynomial-time algorithm for the CLIQUE decision problem: given a graph $G$ and integer $k$, run the BREAKFAST algorithm to find the largest clique, and check whether the clique returned is of size $\geq k$. Because CLIQUE is NP-complete, this would imply that all NP problems can be solved in polynomial time, i.e., that $P = NP$. 