Problem 1. [True or false] (12 points)

(a) **True** or **False**: The propositions $P \implies Q$ and $Q \implies P$ are logically equivalent.

(Converse error.)

(b) **True** or **False**: The proposition $Q \lor P \lor (P \implies Q)$ is guaranteed to be true, no matter what the truth values of $P$ and $Q$ may be.

(Equivalent to $Q \lor P \lor (\neg P \lor Q)$.)

(c) **True** or **False**: If we are told that the proposition $P$ is true, we are entitled to conclude that $(P \implies Q) \implies P$ must also be true.

(Try $Q = false$.)

(d) **True** or **False**: $\forall n \in \mathbb{N} . (x^2 - x = 0) \lor (x \geq 2)$.

(True for $x = 0, 1, 2, \ldots$)

(e) **True** or **False**: $\forall x \in \mathbb{N} . \exists y \in \mathbb{N} . \forall z \in \mathbb{N} . x + y = z$.

(Counterexample: $z = x + y + 1$.)

(f) **True** or **False**: $\forall x \in \mathbb{N} . \exists y \in \mathbb{N} . x^2 = y$.

($y$ is allowed to depend on $x$, since it appears on the inside.)

(g) **True** or **False**: $(\forall x \in \mathbb{N} . \exists y \in \mathbb{N} . x < y) \equiv (\exists y \in \mathbb{N} . \forall x \in \mathbb{N} . x < y)$.

(Not logically equivalent: the left-hand side is true, but the right-hand side is false.)

(h) **True** or **False**: Let $P(n)$ denote the proposition that $1 + 2 + \cdots + n + (n+1) = n(n+1)/2$. Then $\forall n \in \mathbb{N} . P(n) \implies P(n+1)$ is true.

($P(n) = false$ for all $n \in \mathbb{N}$, so the implication is of the form “false $\implies$ false”.)

(i) **True** or **False**: Define the relation $\prec$ so that $n \prec m$ iff $n$ divides $m$ and $n < m$. Then the relation $\prec$ is a well-ordering on $\mathbb{N}$.

(Any subset of $\mathbb{N}$ has a minimal element under $\prec$: its smallest element will be minimal.)

(j) **True** or **False**: Any 2-party cake-cutting protocol that is fair is also envy-free.

(If Alice receives $x \geq \frac{1}{2}$ of the cake, then Bob receives $1 - x \leq \frac{1}{2} \leq x$.)

(k) **True** or **False**: In the stable marriage problem, if $M$ denotes the male-optimal matching, then there exists a girl who does not get her optimal boy in $M$ (i.e., her mate in $M$ is not her optimal boy).

(If $b_i$ lists $g_i$ as his top choice, and vice versa, then there is only one stable pairing; it is simultaneously male- and female-optimal.)
(I) True or False. Suppose that we have \( n \) boys, \( b_1, \ldots, b_n \), and \( n \) girls, \( g_1, \ldots, g_n \). Assume that \( g_1 \) lists \( b_1 \) as her top choice, \( g_2 \) lists \( b_2 \) as her top choice, and so on, so that \( b_i \) appears at the front of \( g_i \)'s preference list for all \( i \). If we run the Traditional Marriage Algorithm, then the resulting matching is guaranteed to match \( g_i \) to \( b_i \) for all \( i \).

(\textit{Let } \( n = 2 \), \( b_1 \) \textit{likes } \( g_2 \) \textit{best}, \( b_2 \) \textit{likes } \( g_1 \) \textit{best}; \textit{then TMA gives the boys their top choices, but not the girls.})

Problem 2. [A peculiar sequence] (6 points)

Define \( u_0 = u_1 = 1 \), and define \( u_n = (n-1)u_{n-2} \) for \( n = 2, 3, 4, \ldots \). By convention, \( 0! = 1 \).

Prove that \( u_{n+1}u_n = n! \) for all \( n \in \mathbb{N} \).

Proof by simple induction on \( n \), with the predicate \( P(n) = "u_{n+1}u_n = n!". \)

Base case: For \( n = 0 \), we have \( u_1u_0 = 1 = 0! \).

Inductive step: Assume \( u_{n+1}u_n = n! \). Then, by the definition of \( u_{n+2} \), we have \( u_{n+2} = (n+1)u_n \) (since \( n+2 \geq 2 \)). Moreover,

\[
\begin{align*}
    u_{n+2}u_{n+1} & = (n+1)u_{n-1}u_n \\
    & = (n+1) \cdot n! \\
    & = (n+1)! 
\end{align*}
\]

(by the inductive hypothesis)

Therefore \( P(n) \implies P(n+1) \) for all \( n \in \mathbb{N} \), so the claim follows by induction.
Problem 3. [Recognizing complete binary trees] (6 points)

A binary tree is a tree with the property that every internal node (i.e., every non-leaf node) has exactly two children. A binary tree is called complete if all its leaves are at the same depth.

Assume that if $T$ is a tree, then the helper function $(\text{leaf? } T)$ returns true if $T$ has no children. Also, if $T$ is a tree with children, then $(\text{left } T)$ returns the left subtree of $T$ and $(\text{right } T)$ returns the right subtree of $T$. Consider the following algorithm:

\begin{verbatim}
(define (complete? T)
  (if (leaf? T)
      #t
      (and (complete? (left T)) (complete? (right T))) )
)
\end{verbatim}

Here is an attempt to prove this algorithm correct.

**Theorem 0.1:** For all trees $T$, if $(\text{complete? } T)$ returns true, then there exists $k \in \mathbb{N}$ such that every leaf is at distance $k$ from the root of $T$.

**Proof:** Use proof by structural induction. Base case: Suppose the tree $T$ has no children, so that $(\text{leaf? } T)$ returns true. Then $(\text{complete? } T)$ also returns true, and moreover every leaf is at distance 0 from the root (i.e., $k = 0$), so the implication is true in this case.

Inductive step: Assume the claim is true for trees $T_L$ and $T_R$. Suppose the tree $T$ is constructed so that the left subtree of the root is $T_L$, and the right subtree is $T_R$. Assume $(\text{complete? } T)$ returns true. By the definition of $(\text{complete? } T)$, this means that $(\text{complete? } T_L)$ and $(\text{complete? } T_R)$ must both have returned true. Then, by the induction hypothesis, there is some $k$ so that every leaf of $T_L$ is at distance $k$ from the root of $T_L$. Similarly, there is some $k$ so that every leaf of $T_R$ is at distance $k$ from the root of $T_R$. In both cases, the leaf is at distance $k + 1$ from the root of $T$ (by the way that $T$ was constructed from $T_L$ and $T_R$). Also, every leaf of $T$ falls into one of these two cases. Consequently, we have proven that if $(\text{complete? } T)$ returns true, then there exists $k' \in \mathbb{N}$ so that every leaf of $T$ is at distance $k'$ from the root of $T$ (in fact, $k' = k + 1$). □

Please tell us whether this proof is valid. In particular, answer each of the following questions with “Yes” or “No”, and explain your answer:

(a) Is the use of structural induction appropriate?

Yes. Structural induction is a fine way to prove that something is true for all binary trees.

(b) Is the proof of the base case ok?

Yes. The base case is indeed a tree with just one node.

(c) Is the proof of the inductive step ok?

No! It is true that every leaf of $T_L$ is at distance $k_L$ from the root of $T$, for some $k_L$, and likewise every leaf of $T_R$ is at distance $k_R$ from the root—but there is no guarantee that $k_L = k_R$.

(d) Bottom line: Is the proof valid?

No. The inductive step is wrong. In fact, the claim is false: $(\text{complete? } T)$ might return true on an input that is not a complete tree.
Problem 4. [Working with expressions some more] (8 points)

This question involves expressions generated by the following rules. (Note that rule 2. is slightly different from what you saw on a previous quiz.)

1. A single digit is an expression.
2. If \( E_1 \) and \( E_2 \) are expressions, then \( E_1 E_2 \) is an expression.
3. If \( E \) is an expression, then \( (E) \) is an expression.

Prove that no expression generated by the three rules above contains an open parenthesis immediately followed by a close parenthesis, i.e. “()”.

Proof by strong induction on the number of applications of the rules. Let \( P(k) = \text{“for every expression } E \text{ generated using exactly } k \text{ applications of the above rules, } E \text{ contains no (), } E \text{ does not start with }, \text{ and } E \text{ is not empty”} \)

Base case: For \( k = 1 \), the only way to get an expression is by application of rule 1. A single digit contains no parentheses, so it contains (), does not start with ), does not end with (, and is not empty.

Inductive step: Suppose \( P(1) \land \cdots \land P(k) \) is true, i.e., every expression generated using at most \( k \) rule-applications has no (), doesn’t start with ), end with (, and isn’t empty. We will prove that \( P(k+1) \) is true. Let \( E \) denote any expression generated using exactly \( k+1 \) rule-applications. There are three cases, divided into which rule was used in the \( k+1 \)th rule-application:

1. If the last rule was Rule 1.: \( E \) is a single digit, so it contains no (), doesn’t start with ) or end with (, and is non-empty.
2. If the last rule was Rule 2.: \( E = E_1 E_2 \) for some expressions \( E_1 \) and \( E_2 \). \( E_1 \) must have been generated using at most \( k \) rule-applications and thus (by the inductive hypothesis) has no (), doesn’t end in (, and is non-empty. Likewise, \( E_2 \) has no (), doesn’t start with ), and is non-empty. It follows that the concatenation \( E_1 E_2 \) has no (), by considering where a ( could occur (anywhere in \( E_1 \) except the last symbol, and anywhere in \( E_2 \)) and using the fact that neither \( E_1 \) nor \( E_2 \) has ). Moreover, \( E_1 E_2 \) doesn’t start with ) (since \( E_1 \) doesn’t, and is non-empty), doesn’t end with ( (since \( E_2 \) doesn’t, and is non-empty), and \( E_1 E_2 \) is non-empty (since \( E_1 \) is non-empty).
3. If the last rule was Rule 3.: \( E = (E_1) \), for some expression \( E_1 \). \( E_1 \) must have been generated using at most \( k \) rule-applications and thus (by the inductive hypothesis) has no (), doesn’t end in ( or start with ), and is non-empty. It follows that \( (E_1) \) has no (): the first ( in \( (E_1) \) is not immediately followed by a ) (since \( E_1 \) is non-empty); and any other ( in \( (E_1) \) is not immediately followed by a ), since such a ( must occur in \( E_1 \), not at the end of \( E_1 \), and thus cannot be immediately followed by a ). Moreover, \( (E_1) \) does not start with ) or end with (, and is not empty.

Alternate approach: You could have used \( P(k) = \text{“no expression generated with } k \text{ rule-applications contains a ()”} \), and then proved a separate Lemma stating that no expression starts with ), ends with (, or is empty. Such a Lemma could also be proved using strong induction.
Common errors:

• Many answers started from an example of something that would be bad, and then tried to prove it couldn’t happen. (What about other bad things?)

Some people tried to argue that () is not an expression, and stopped there. But that is not enough: you had to show that no expression contained () as a substring.

• Converse errors were common. Example: “Suppose \( E = (E_1) \) is an expression. Then \( E_1 \) must be an expression.” — wrong. (Counterexample: \( E = (1)(2) \).)

Some answers tried starting with an expression and ‘breaking it down’, but such proofs usually failed to justify why you can ‘break an expression down’ and often succumbed to converse errors.

• Many people failed to state what induction predicate \( P(n) \) they were using, or what kind of induction they were using (simple? strong? structural? infinite descent?).

• If you use induction on the number of rules, you need to use strong induction. Suppose \( E \) was generated using \( k + 1 \) rule-applications, and \( E = E_1E_2 \) (by Rule 2.). You need to take into account the possibility that \( E_1 \) and \( E_2 \) were generated with strictly less than \( k \) rule-applications.

• Many proofs did a case analysis of how some expression could arise, but failed to check that the cases covered all possibilities. Example: “Let \( E \) be an expression of size \( n + 1 \). The only way to get \( E \) out of an expression of size \( n \) is by . . .” (What if there is some other way to get \( E \)? For instance, what if \( E \) is built out of an expression \( E_1 \) of size 2 and an expression \( E_2 \) of size \( n - 2 \)?)

• Some people noticed that we need the fact that no expression starts with ), ends with (, or is empty—but then asserted this without proof. Recognizing this subtlety is good, but you also need to prove that this invariant holds (e.g., by induction).

• A common error was to assert that the induction hypothesis applies without explaining why. A rarer error was to assume what was to be proven. Example: “Let \( P(k) = \) ‘no expression with \( k \) digits contains a (‘). To show: \( P(k) \implies P(k+1) \). Consider any expression \( E = (E_1) \) with \( k + 1 \) digits. By the induction hypothesis, \( E_1 \) contains no (), therefore...” — bogus. \( E_1 \) has \( k + 1 \) digits, so you need to assume \( P(k+1) \) to prove \( P(k+1) \).

• Example: “Rule 3. is the only rule that adds parentheses, so no expression can start with ) [or, contain (, or . . .].” (Fails to consider the possibility that some other rule might introduce the forbidden string without adding any new parentheses. For instance, maybe Rule 2. takes a ) and moves it to the front. It doesn’t, but this could use justification.)

• Some people tried using predicates that make induction too hard. Example: “Let \( n = \) the number of applications of Rule 3.” (Base case becomes non-trivial: there are infinitely many expressions formed without any use of Rule 3.) Example: “Let \( n = \) the number of digits.” (The induction step becomes too hard, because we could apply Rule 3. many times without changing \( n \).)

Stylistic advice on good proofs:

• All variables in a proof should be immutable. Don’t try changing the meaning of a variable. Example: “We start with \( E \), and we add parentheses surrounding it, and afterwards \( E \) still contains no (‘).”
• Be wary of type errors. Example: “The expression $P(n)$ has no ().” — wrong. ($P(n)$ is a boolean proposition, not an expression.)

• If you haven’t used the induction hypothesis anywhere in your proof, that is suspicious.
Problem 5. [A variant on merge sort] (8 points)

Let $F_k$ denote the $k$th Fibonacci number. Reminder: the Fibonacci numbers are defined by $F_0 = F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for $k \geq 2$. Consider the following variation on merge sort, which assumes that the number of elements in its list argument $L$ is a Fibonacci number $F_k$.

```
algorithm FibMergesort(L)
    if L contains only 1 element, then return L
    else
        divide L into $L_1$ (the first $F_{k-1}$ items) and $L_2$ (the remaining $F_{k-2}$ items)
        sorted$L_1 := \text{FibMergesort}(L_1)$
        sorted$L_2 := \text{FibMergesort}(L_2)$
        sortedL := Merge(sorted$L_1, sorted$L_2$)
        return sortedL
```

Assuming that the “divide” step in FibMergesort takes constant time (no comparisons) and Merge behaves as described in the lecture notes, identify which of the following expressions most closely matches the total number of comparisons performed by FibMergesort when initially given a list of $F_k$ elements.

(a) $O(k \log k)$
(b) $O(k^2)$
(c) $O(k F_k)$
(d) $O(F_k \log k)$
(e) $O(F_k^2)$

The depth of the recursion tree is $k - 1$. The total number of comparisons done at each level of the tree is at most $F_k$, since the size of the lists at each level sum to $F_k$, and the number of comparisons used by Merge is no more than the total list size. (In fact, we do strictly less than $F_k$ comparisons at the bottom $k/2$ levels, but that’s OK.) Thus the total number of comparisons is the number of levels times the number of comparisons done per level, or $O(k F_k)$.

Alternate answer: Let $T(k)$ = the number of comparisons performed by FibMergesort on an input list of size $F_k$. Then $T(k) = T(k - 1) + T(k - 2) + F_{k-1}$ (since Merge uses $F_{k-1}$ comparisons to merge two lists of total size $F_k$), and $T(1) = T(0) = 0$.

Claim: $T(k) \leq k F_k$ for all $k$.

Proof: By strong induction on $k$. Base cases: $T(0) = 0 \leq 0 \cdot 1$ and $T(1) = 0 \leq 1 \cdot 1$. Inductive step: Assume it is true for $k - 1$ and $k - 2$. Then

$$T(k) = T(k - 1) + T(k - 2) + F_{k-1}$$
$$\leq (k - 1)F_{k-1} + (k - 2)F_{k-2} + F_{k-1}$$
$$\leq (k - 1)F_{k-1} + (k - 1)F_{k-2} + F_{k-1}$$
$$\leq (k - 1)F_k + F_k \leq k F_k.$$