7.11 By definition, it is sufficient to construct a (deterministic) polynomial time **verifier** for ISO. (Alternatively, one can construct a nondeterministic polynomial time **decider**.) We denote a graph as \((V, E)\), where \(V\) is the vertex set and \(E\) is the edge set. Consider a permutation as the certificate:

\[
V = \text{"On the input } (G, H, c)\text{, where } c \text{ is a permutation on } |V_G| \text{ elements, and } G = (V_G, E_G) \text{ and } H = (V_H, E_H) \text{ are graphs.}
\]

1. Test whether \(c\) is a permutation from \(V_G\) to \(V_H\).
2. Test whether \(|V_G| = |V_H|\) and \(|E_G| = |E_H|\).
3. Test whether \(\forall v_i \neq v_j \in V_G, (v_i, v_j) \in E_G \iff (c(v_i), c(v_j)) \in E_H\).
4. If all pass, accept; otherwise, reject.”

(Time complexity:) In the step 1 the running time to check the one-to-one and onto properties is \(O(|V_G|+|V_H|)\). In the step 2, the running time is less than \(\max\{ |V_G|, |V_H|, |E_G|, |E_H| \}\). Suppose \(|V_G| = |V_H|\) and denote \(n = |V_G| = |V_H|\), then the step 3 needs time less than \(O(n^2)\) since \(|E| \leq n(n-1)/2 < n^2\). The total running time is \(O(n^2)\). Hence \(V\) is a polynomial time verifier, which means that \(ISO \in NP\).

7.12 Because \(b\) is a binary integer, let \(b = b_1b_2 \cdots b_{n-1}b_n\). (So the decimal representation of \(b\) is \(\sum_{k=1}^{n} b_{n-k+1}2^{k-1}\).) By the hint, we can observe that \(a^{(100)} = a^2\) and \(a^{(1000)} = (a^2)^2\). Hence we can construct the following algorithm to decide **MODEXP**:

\[
A = \text{"On the input } (a, b, c, p)\text{, where } a, b, c \text{ and } p \text{ are binary integers.}
\]

1. Let \(T = 1\) (and \(n = \lfloor \log_2 b \rfloor \)).
2. For \(i = 1\) to \(n\),
   
   \[
   \begin{align*}
   &\text{if } b_i = 1, T = (a(T^2) \pmod{p}), \\
   &\text{if } b_i = 0, T = (T^2 \pmod{p});
   \end{align*}
   \]
3. Return \(T \pmod{p}\).
4. If \(T = c \pmod{p}\), accept; otherwise, reject.”

(Time complexity:) Assume that \(a, b, c\) and \(p\) are at most \(m\) bits (so \(n \leq m\)). It is known that two \(m\)-bit numbers cost \(O(m^2)\) unit time to do multiplication and division (and hence modular), so each \(i\) costs \(O(m^2)\) time. The total cost of the for loop is \(O(m^2) \times n = O(m^3)\), which is the dominant cost of all steps. The time complexity of the algorithm \(A\) is a polynomial in the length of its input.

7.14 Consider any language \(A \in P\). We show that \(A^*\) also belongs to \(P\) by constructing a polynomial time algorithm to decide \(A^*\).

For any input string \(w = a_1a_2...a_n\), let \(w_i\) and \(w_{i,j}\) be the substring \(a_1a_2...a_i\) and \(a_ia_{i+1}...a_j\). For convenience we define \(w_0 = \epsilon\). And define functions

\[
f(i) = \begin{cases} 
1, & \text{if } w_i \in A^*; \\
0, & \text{if } w_i \notin A^*.
\end{cases}
\]
\[ g(i, j) = \begin{cases} 1, & \text{if } w_{i,j} \in A; \\ 0, & \text{if } w_{i,j} \notin A. \end{cases} \]

By the definition, \( f(i) \) itself has a recurrence relation:

\[ f(i) = \begin{cases} 1, & \text{if } i = 0; \\ 1, & \text{if there exists some index } 0 < j \leq i \text{ s.t. } g(j, i) = 1 \text{ and } f(j - 1) = 1; \\ 0, & \text{Otherwise.} \end{cases} \]

Since the value of \( f(i) \) depends on \( f(j) \) for only \( j < i \), we have an algorithm that it compute \( f(i) \) with dynamic programming in increasing order of \( i \). For each \( f(i) \) it simply enumerates all possible \( j \) to check the value of \( g(j, i) \) and \( f(j - 1) \). Because \( A \) belongs to \( P \), we can compute the value of \( g(j, i) \) in polynomial time so the algorithm entirely takes polynomial time computation.

7.20.a To show \( SPATH \in P \), we can construct a TM \( T \) which can decide \( SPATH \) in polynomial time.

\[ T = \text{"on input } < G, a, b, k > \text{"} \]

1. Use BFS starting from \( a \), and mark the searched nodes with their depths.
2. If the node \( b \) is marked and its depth is less than or equal to \( k \), accept; otherwise, reject.

(Time complexity:) In the step 1, BFS traverses every node at most once, so its time complexity is \( O(|V| + |E|) \). The step 2 to can be finish in one comparison, so \( T \) can decide \( SPATH \) in polynomial time \( (O(|V| + |E|)) \). We can conclude \( SPATH \in P \).

7.20.b To show \( LPATH \) is NP-complete, we prove \( LPATH \in NP \) first, and then show that \( UHAMPATH \leq_p LPATH \). We can construct a verifier \( V \) for \( LPATH \):

\[ V = \text{"on input } < G, a, b, k, C > \text{, where } C \text{ is a path.} \]

1. Check \( C \) is a non-repeated sequence of nodes in \( G \).
2. Check the first term of \( C \) is \( a \) and the last is \( b \).
3. Check the length of \( C \) is larger than or equal to \( k \).
4. If \( C \) satisfies the conditions 1 to 3, accept; otherwise, reject.

This verifier \( V \) can finish in \( O(|C|) \) where \( |C| \) is the length of \( C \), so \( LPATH \in NP \).

Now we show that \( UHAMPATH \leq_p LPATH \). Consider an instance \( < G, a, b > \) of \( UHAMPATH \) problem , where \( G = (V, E) \) is a graph with assigned starting node \( a \) and ending node \( b \). The mapping copy \( < G, a, b > \) and set \( k = |V| - 1 \), then \( < G, a, b, k > \) is an instance of \( LPATH \). It can be finished in polynomial time \( (O(|V| + |E|)) \). We need to prove \( < G, a, b > \in UHAMPATH \iff < G, a, b, k > \in LPATH \):

(\( \Rightarrow \)) If \( < G, a, b > \in UHAMPATH \), then \( G \) has a hamiltonian path from \( a \) to \( b \). It must be a simple path that goes through every node exactly once, which implies that the length is \( |V| - 1 = k \). So \( < G, a, b > \in LPATH \).

(\( \Leftarrow \)) If \( < G, a, b > \in LPATH \), there exists a simple path from \( a \) to \( b \) with length \( k = |V| - 1 \). Because the graph \( G \) only has \( k + 1 \) nodes, so this simple path must pass through all of nodes in \( G \) exactly once. So this simple path must be a hamiltonian path. It implies that \( < G, a, b > \in UHAMPATH \).
7.24.a Because with a ∼-assignment there are at least one literal be set true and false in each clause, the negation of it would also have at least one true literal that was be set false originally and at least one false literal for same reason in each clause.

7.24.b Obviously the reduction could be computed in polynomial time. To show the reduction mapping is correct, consider any 3cnt-formula φ and its reduction φ′. If τ is a satisfying assignment of φ, we can extend it to obtain a ∼-assignment of φ′, called τ′. Consider the assignment values of the three literals y₁, y₂, y₃ in clause cᵢ of φ set by τ. If y₁ or y₂ is true, then we set zᵢ as false, otherwise we set zᵢ as true in τ′. And let b be false. Then τ′ is a ∼-assignment for φ′ since τ is a satisfying assignment for φ that there is at least one literal be set true in each clause. Conversely, if there is a ∼-assignment τ′ for φ′. Observe that if (y₁ ∨ y₂ ∨ zᵢ) and (zᵢ ∨ y₃ ∨ b) are ∼-satisfied, then at least one of yᵢ must be true. If y₁, y₂, y₃ are all false, then zᵢ and b must be true. However, by (a), we know we can negate the assignments, such that at least one yᵢ’s is true. Also note that in a satisfying ∼-assignment, "b = false" implies at least one yᵢ’s must be true.

So φ is satisfiable if and only if φ′ has a ∼-assignment.

7.24.c Since we can verify whether an assignment is a ∼-assignment for a 3-cnf-formula in polynomial time, ∼ SAT is obviously in NP. By the result of (b), we know it is also NP-Hard, so ∼ SAT is NP-complete.