We will show that $ISO \in NP$ using a polynomial time verifier, the verifier takes as input $(G_1, G_2, w)$, where $G_1, G_2$ are the two (possibly) isomorphic graphs, and $w$ is a possible witness which is a reordering of $G_2$.

You can think of $w$ as a map between nodes in $G_2$ and corresponding nodes in $G_1$, or as a map that renames nodes in $G_2$ to new names, such that if we rename $G_2$ we should get $G_1$. We require that $w$ may only map a node to a node in $G_1$, and may not map two nodes to the same value. We also require that $G_1, G_2$ have the same number of nodes (otherwise they are clearly not isomorphic).

To check if $w$ is an actual witness, we will use it to reorder/rename $G_2$, and then check that $G_1$ and the reordered $G_2$ are the same graph. The verifier executes these steps:

1. For every node $n \in G_2$. Rename $n$ to $w[n]$ (the value mapped to $n$ by $w$). Call the new renamed graph $G_2'$.

2. For every $i \in [1, |G_1|]$, check that $\text{neighbors}_{G_1}(n_i) = \text{neighbors}_{G_2'}(n_i)$. In other words, $n_i$ is connected to the exact same set of nodes in both $G_1$ and $G_2'$.

3. If the above condition holds for all $i$, accept. Otherwise, reject.

The verifier runs in polynomial time. Step one is repeated for every node, therefore it can be carried out in polynomial time. The second step needs $O(|G_1|^2)$, since it is repeated for every node (at most), and a node can be (at most) connected to all other nodes.

If the verifier accepts, it means that $w$ correctly reordered $G_2$ such that it became identical to $G_1$. If the verifier rejects, then at least one node had different neighbors in the two graphs (or the graphs are of unequal size), therefore $w$ does not correctly reorder $G_2$. 

---

1
2. Define SuperSat to be \( \{ F \mid F \text{ is a propositional formula and } F \text{ has at least 3 satisfying assignments} \} \). Prove that SuperSat is NP complete.

**Solution:**

*Membership in NP:* We can prove SuperSAT \( \in NP \) using a polynomial time verifier. The polynomial time verifier takes as input \((F, w)\) where \(F\) is a propositional formula, and \(w\) is a possible witness made out of three assignments \(w = (w_1, w_2, w_3)\).

The verifier checks that these assignments are not equal, and that each of them is a satisfying assignment for \(F\), by replacing the variables of \(F\) with their values according to the assignment, and checking that the evaluation of \(F\) is true.

Evaluating every one of those assignments can be done in linear time. Checking that the assignments are different also needs linear time (string comparison). So the verifier runs in linear (polynomial) time.

*NP-hardness:* We will reduce (in polynomial time) SAT to SuperSAT. Given a Boolean (propositional) formula \(F\), we will construct a new propositional formula \(F'\) such that \(F' \in \text{SuperSat}\) if and only if \(F \in \text{SAT}\).

Let \(x_{n1}, x_{n2}\) be new unique variables that do not appear in \(F\) (have new unique names). Let \(F' = F \land (x_{n1} \lor x_{n2})\).

If \(F\) is satisfiable, then it has at least one satisfying assignment, we can extend this satisfying assignment into three satisfying assignments for \(F'\) by adding each of these three combinations to that assignment: \((x_{n1} = T, x_{n2} = T), (x_{n1} = F, x_{n2} = T), (x_{n1} = T, x_{n2} = F)\). Every one of these extended assignments satisfies both \(F\) and \((x_{n1} \lor x_{n2})\) by construction. Therefore \(F' \in \text{SuperSAT}\).

If \(F\) is unsatisfiable, then there is no assignment that can satisfy \(F\). Which means that \(F'\) has no satisfying assignments, since \(F'\) is made up from a conjunction (and) that includes \(F\) (which is unsatisfiable). Therefore \(F' \notin \text{SuperSAT}\).

Constructing \(F'\) can be done in polynomial time (string concatenation), so this reduction can be carried out in polynomial time. Therefore SuperSAT is NP-complete.

3. Assume that P=NP. Using this assumption prove that, given a formula \(F\), you can produce a truth assignment which satisfies \(F\) in polynomial time, if one exists.

Note: The P=NP assumption only tells you that you can DECIDE whether a formula is satisfiable. This problem asks you to produce such a satisfying assignment.

**Solution:**

Since we assume \(P = NP\), then there exists a polynomial-time deterministic Turing machine \(M\) that decides the language \(\text{SAT} = \{ F \mid F \text{ is a propositional formula that is satisfiable } \}\). We will build an
algorithm/Turing machine that uses \( M \) repeatedly to find a satisfying assignment.

Let \( F \) be the input propositional formula. Let \( X_1, X_2, ..., X_n \) be all the variables that appear in \( F \). Consider the following algorithm:

1. Run \( M \) on \( F \), if \( M \) rejects, then there is no satisfying assignment, if \( M \) accepts move to the next step.
2. Initialize \( A = [\], i = 1 \).
3. Construct \( F' = F[X_i \mapsto T] \). In other words, \( F' \) is a copy of \( F \) where every occurrence of \( X_1 \) is set to true.
4. Run \( M \) on \( F' \), if \( M \) accepts, then \( F' \) is satisfiable, and \( X_i = T \) is part of some satisfying assignment. Therefore copy \( F' \) into \( F \) and add \( X_i = T \) to \( A \).
5. If \( M \) rejected, then we chose the wrong value for \( X_i \) which made \( F \) unsatisfiable. The only other option is for \( X_i \) to be false. Therefore make \( F = F[X_i \mapsto F] \) and add \( X_i = F \) to \( A \).
6. Increment \( i \) by one, if \( i < n \) repeat step 3.

The algorithm above runs in polynomial time. Steps 3 to 5 are repeated for \( i = 1, ..., n \). Running \( M \) once requires polynomial time. Replacing variables with values can be done in linear time. All other operations can be done in constant time. In total, the algorithm needs \( O(n \times \text{time}(M)) \) where \( \text{time}(M) \) is polynomial.

The algorithm finds a satisfying assignment if one exists. If \( F \) is satisfiable, then for every \( i \leq n \) a satisfying assignment exists with \( X_i \) set to either True or False. The algorithm builds a satisfying assignment one variable at a time. The algorithm tries to set \( X_i \) to True if \( M \) tells us that this will keep the formula satisfiable. Otherwise \( X_i \) must be false.

4. The Subset Sum problem is defined on page 269 (Section 7.3) of the textbook.
   Show that Subset Sum is in linear space (linear space = Space \( O(n) \)).

**Solution:**

We will show that there exists a deterministic Turing machine that decides SubsetSum in linear space. Notice that there are no restrictions on the run-time on the machine. In fact, our machine will run in exponential time. Remember that SubsetSum is NP-Complete, and therefore unlikely to be solved deterministically in polynomial time.

The Turing machine is very simple:
“On input \(< S, k>\), where \( S \) is a set containing numbers, and \( k \) is the desired sum:
1. For every subset \( a \subseteq S \), check if \( a = k \).
2. If one such subset is found, accept. Otherwise, reject.”
This machine clearly runs in exponential time, since it needs to try all possible subsets in the worst-case, and there are $2^{|S|}$ such subsets.

However, this machine can run in linear space. The machine needs space to store one subset at a time, as well as space to store the sum. Any subset of $S$ cannot be larger than $S$, therefore we do not need more than linear space to store any single subset.

All that is left to do is show a way for us to move from one subset to a new subset, without repeating or missing any subset. We can do this using a bit-mask (this is a very popular technique). Let $a$ be represented as a binary number with $|S|$ digits. The $i$-th digit of $a$ indicates if the $i$th element of $S$ is in the subset or not (depending on whether the digit is 1 or 0). Initially, we set $a = 0...0$ to represent the empty subset. After every iteration we increment $a$ by 1. we stop when $a = 1...1$ which represent the subset equal to the entire set.

We only need linear space to store the bitmask, as well as space to store one integer (the sum of the subset represented by the current bitmask). The largest possible sum is $|S| \times l$ where $l = \max(S)$. Therefore we need space in $O(\log(|S| \times l)) \in O(|S|)$ to store the sum. Therefore, we need $O(|S|)$ space in total.

5. A corollary to Theorem 8.5 of the textbook (Savitch’s theorem) says that $\text{PSPACE} = \text{NPSPACE}$. You can use these results in your answers to i. and ii below.

i. Briefly state why $\text{NP}$ is contained in $\text{PSPACE}$.

Solution:

We know that $\text{NP} \subseteq \text{NPSPACE}$. A language in NP is decided by a polynomial time non deterministic Turing machine. Since this machine may only run for a polynomial time, it can only allocate and use a polynomial amount of memory.

By Savitch’s Theorem, we know that $\text{NPSPACE} = \text{PSPACE}$, therefore $\text{NP} \subseteq \text{PSPACE}$.

ii. $\text{PSPACE}$ completeness is defined just like NP complete, for details see section 8.3 of the textbook.

Prove that if some PSPACE-complete set $C$ is in NP then $\text{NP} = \text{PSPACE}$.

Solution:

We already know from the previous part that $\text{NP} \subseteq \text{PSPACE}$. All we need to do is prove that $\text{PSPACE} \subseteq \text{NP}$ under the given assumptions.
Let $C$ be a PSPACE-Complete set, and $C \in NP$. Since $C \in NP$, then there exists a NTM $M_C$ that decides $C$ in polynomial time. By definition of PSPACE-Completeness, we know that for all $A \in PSPACE : A \leq_P^n C$. Which means that for every $A$, there exists some function $f_A$ such that $w \in A \equiv f_A(w) \in C$, and $f_A$ can be computed in polynomial time.

Given any set/problem $A \in PSPACE$, we will show that $A \in NP$ by constructing NTM $M_A$ that decides $A$ in polynomial time. This is sufficient to show that $PSPACE \subseteq NP$.

$M_A =$ “On input $w$:

1. Compute $g = f_A(w)$.
2. Simulate NTM $M_C$ on $g$. In every computation path where $M_C$ accepts, accept. In other paths, reject.”

$M_A$ runs in polynomial time, since both steps run in polynomial time. Further more $M_A$ decides $A$, since it accepts $w$ if and only if $M_C$ accepts $f_A(w)$, which can only happen if $f_A(w) \in C$, which means that $w \in A$ (by the definition of reduction above).