Assignment 4

Question 1

To prove that f is not **NP**-hard under $\leq$, a problem will be appropriately chosen, that is in **NP**, and it will be shown that that problem cannot be reduced to f. In this case, the chosen problem will be CIRCUIT-SAT, which is already proven to be **NP**-complete. So, it will be shown that CIRCUIT-SAT $\nleq f$, which proves that f is not **NP**-hard.

CIRCUIT-SAT is the decision problem where given a Boolean circuit, does there exist an assignment of its inputs that makes the output true. In other words, does there exist inputs of 0 and 1 to the circuit that evaluates to 1. To solve this problem, an algorithm has to try every possible combination of inputs and evaluate the circuit each time to see if it outputs 1. So, given an input of assignments, although the computation of the circuit to determine the output can be done in constant time, the determination of the inputs is done in exponential time, as the algorithm would have to try every possible combination in the worst-case.

With that being said, now attempt to reduce CIRCUIT-SAT to f via $\leq$. It is stated that the function f has the following property: $f(x) = true$ for every $x \in \{0,1\}^*$, where $\{0,1\}^*$ is the set of all bit-strings. So, in this case, since $x \in \{0,1\}^*$, then this means that the input to f is always a valid bit-string, thus $f(x)$ will always evaluate to $true$. This reveals the discrepancy between these two problems: CIRCUIT-SAT will output either $true$ or $false$, whereas f always outputs $true$. In addition, CIRCUIT-SAT must compute its output based on its input, meaning it must process and determine whether the input is valid, which leads to its exponential time as mentioned earlier, whereas f will always evaluate to $true$, leading to a constant time of $O(1)$, putting it in the complexity class **P**.

Since f runs in constant time, whereas CIRCUIT-SAT runs in exponential time, it is not possible to say that CIRCUIT-SAT can reduce to f. Another perspective: take the bit-string input x of f, break them up into individual bits, and assign them to some circuit C as the inputs. In this case, $f(x)$ will evaluate to $true$ no matter what the setup of the circuit is, whereas CIRCUIT-SAT may evaluate to $true$ or $false$ depending on the circuit. Thus, CIRCUIT-SAT cannot reduce to f because not only is it more computationally complex than f, but it also could result in different outputs for the same input. Therefore, CIRCUIT-SAT $\nleq f$, hence f is not **NP**-hard.

Assignment 4

Question 2

To prove that the k connected components decision problem is **P**-complete under $\leq$, two properties will be shown:

(i) This problem is itself $\in$ **P**.

(ii) Every problem $\in$ **P** reduces to this problem.

To show that this problem is indeed in **P**, a deterministic polynomial-time algorithm will be proposed that solves the problem. To show that it is **P**-hard, a reduction will be carried out from some other problem in **P** to this problem. Proving these two together is enough evidence to claim that this problem is indeed **P**-complete.

To prove (i), the following is a deterministic polynomial-time algorithm (written in C++) that solves the k connected components problem:

```
void dfs(int v, vector<bool> visited, vector<vector<int>> adj) {
    visited[v] = true;
    for (auto it = adj[v].begin(); it != adj[v].end(); ++it) {
        if (!visited[*it]) {
            dfs(*it, visited, adj);
        }
    }
}

int countConnectedComponents(vector<bool> visited,
vector<vector<int>> adj) {
    const int n = visited.size();
    int connectedComponents = 0;

    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i, visited, adj);
            connectedComponents++;
        }
    }

    return connectedComponents;
}
```

```
bool hasKConnectedComponents(const int connectedComponents,
const int k) {
    if (connectedComponents == k) {
        return true;
    }
    return false;
}
```

As mentioned, this algorithm was written in C++ rather than pseudocode to be able to test with various inputs of non-empty undirected graphs G and positive integers k. The algorithm consists of three functions: dfs, $countConnectedComponents$, and $hasKConnectedComponents$. The first function dfs is given a starting vertex, visits it, and then visits every neighbouring vertex that has not yet been visited through a recursive strategy, which results in a depth-first search of the graph. The function $countConnectedComponents$ initially sets all visited nodes to false, as well as a connected components counter to 0, and then for each non-visited node, calls the dfs method and increments the counter. This works because the dfs function naturally visits all the vertices that are connected to the starting vertex, so once that function is finished, it is guaranteed that that is one connected component. Finally, $hasKConnectedComponents$ compares the number of connected components computed in $countConnectedComponents$ to the positive integer k, and if they are equal then return true, else return false.

Analyzing the time efficiency of this algorithm, it performs a depth-first search through the entire graph G, visiting all the nodes once, leading to a time efficiency of $O(|V| + |E|)$, where V are the vertices and E are the edges of the graph G. Thus, this algorithm is linear in time, and in turn, safely upper bounded polynomial in time. Therefore, the k connected components problem is proven to be itself ∈ **P**, and (i) is satisfied.

To prove (ii), let A be any problem in **P**. Now, need to show that A reduces to the k connected components problem with Definition 1 from Lecture 9(c). Once this is achieved, then this proves that the problem is **P**-hard, and together with (i) showing that it is in **P**, then a conclusion can be reached about the problem, namely that it is **P**-complete under ≤.

Since A is assumed to be any problem in **P**, then there exists a polynomial-time algorithm, call it L_A, that given any instance a of A, correctly outputs true or false. Now, one can leverage L_A to reduce A to the k connected components problem. Let B represent the k connected components problem, with L_B being the polynomial-time algorithm discussed above.

Furthermore, let I_A be the set of all instances of A and I_B be the set of all instances of B. Now, suppose A has the following decision problem: given the same positive integer k from B, does there exist a function such that it only outputs true on a certain value equivalent to k, and false otherwise. Let this function be L_A.

Now, given the same inputs as B, the problem A will return true if and only if L_A returns true, and false otherwise. Applying this algorithm to the problem B, this will also result in the same output, namely that it will return true if and only if there are indeed k connected components in the given graph G for problem B. This extends for any instance a of A, where the function L_A will return true for any positive integer k that is satisfied by its desired equivalence. Thus, the following property holds: $a \in I_A$ is true if and only if $L_A(a) \in I_B$ is true. In other words, any instance a of A evaluates to true if and only if the proposed algorithm L_A evaluates to true when applied to problem B, which it indeed does.

Next, it can be proven that the algorithm L_A is indeed polynomial-time computable, which will complete the reduction. The goal of L_A is, given some input, determine whether or not it can achieve some value equivalent to the positive integer k common to both problems. Hence, the algorithm must iterate through the entirety of its given input to determine such a value. However, it only has to do this once, i.e., visit each element once, thus also running at linear time in the size of its input. In other words, it does not have to revisit nodes it has already processed. Therefore, it can be stated that L_A is indeed polynomial-time computable.

At this point, the following claims have been proven: (1) $a \in I_A$ is true if and only if $L_A(a) \in I_B$ is true, and (2) L_A is polynomial-time computable. Thus, it can be said that A reduces to B, or $A \leq B$, where B is the k connected components problem. Since A is represented as any problem in **P**, then this holds in the general case, namely that every problem $\in$ **P** reduces to this problem, therefore the problem is **P**-hard, and (ii) is satisfied.

Since properties (i) and (ii) are both satisfied, then the k connected components decision problem is indeed **P**-complete under $\leq$.

Assignment 4

Question 3

To prove that k-CIRCUIT-SAT $\leq$ 2-CIRCUIT-SAT, utilize Definition 1 from Lecture 9(c): let A represent some decision problem and let B represent some other decision problem. Let I_A be the set of all instances of A and I_B the set of all instances of B. Then, A reduces to B if there exists a function $f: I_A \rightarrow I_B$ such that: (i) $a \in I_A$ is true if and only if $f(a) \in I_B$ is true, and (ii) f is polynomial-time computable. Additionally, exploit Claim 1 from Lecture 9(c), where the notion of reduction, $\leq$, is transitive, meaning if $A \leq B$ and $B \leq C$, then $A \leq C$. In particular, the following procedure will be adopted, namely a chain of reductions:

1. k-CIRCUIT-SAT $\leq$ CNF-SAT
2. CNF-SAT $\leq$ ATMOST-2-CNF-SAT
3. ATMOST-2-CNF-SAT $\leq$ EXACTLY-2-CNF-SAT
4. EXACTLY-2-CNF-SAT $\leq$ 2-CIRCUIT-SAT

Each of these four reductions will be carried out, and using the property of transitivity, this will prove that k-CIRCUIT-SAT $\leq$ 2-CIRCUIT-SAT. To prove (1), given an instance of k-CIRCUIT-SAT, i.e., a Boolean circuit C, label each distinct wire $x_1, \ldots, x_n$. Now, in the output SAT formula, construct a conjunction of clauses: one for the output wire, and one for each gate. Now, the circuit C is satisfiable if and only if the SAT formula is satisfiable, proving (i) of Definition 1. And the SAT formula has a number of clauses which is linear in the number of gates and wires, i.e., linear in the size of C, proving (ii) of Definition 1. Thus, it can be stated that k-CIRCUIT-SAT $\leq$ CNF-SAT, so (1) is proven to be true.

To prove (2), identify that for ATMOST-2-CNF-SAT, each clause has at most 2 literals. Suppose some clauses in an input instance of CNF-SAT has $k > 2$ literals. Let the clause be, $C = l_1 \vee l_2 \vee \ldots \vee l_k$, and introduce $k - 2$ new Boolean variables, $y_1, \ldots, y_{k-2}$. Now, rewrite C as, call it C':

$$(l_1 \vee l_2) \wedge (l_3 \vee y_1) \wedge (l_4 \vee y_2) \wedge \ldots \wedge (l_k \vee y_{k-2})$$

This mapping is linear in time: given a clause of k literals, it ends up with $\Theta(k)$ clauses of 2 literals each, proving (ii) of Definition 1. Seek to show that C is satisfiable if and only if C' is satisfiable. For the "if" direction, assume C' is satisfiable. Consider two cases for a satisfying assignment, call it $f: \{l_1, \ldots, l_k, y_1, \ldots, y_{k-2}\} \rightarrow \{0, 1\}$:

Case 1: $f(l_i) = 0$ for every l_i. This is impossible, thus, it cannot be the case that $f(l_i) = 0$ for every l_i, because if it is, then f is not a satisfying assignment for C'.

Case 2: Some $f(l_i) = 1$. This means f restricted to the l_i's only is a satisfying assignment for $C = l_1 \vee \dots \vee l_k$.

For the "only if" direction, assume $g: \{l_1, \dots, l_k\} \to \{0, 1\}$ is a satisfying assignment for C. Then, some $g(l_i) = 1$. If $i = 1$, then $g(y_1) = \dots = g(y_{k-2}) = 0$ is a satisfying assignment for C'. If $i > 1$, then $g(y_1) = \dots = g(y_{i-1}) = 1$ and $g(y_i) = \dots = g(y_{k-2}) = 0$ is a satisfying assignment for C', proving (i) of Definition 1. Thus, it can be stated that CNF-SAT $\leq$ ATMOST-2-CNF-SAT, so (2) is proven to be true.

To prove (3), given a clause that is one literal only, introduce one new variable y and write $C = l$ as C': $(l \vee y) \wedge (l \vee \neg y)$. Given a clause that comprises two literals only, introduce no new variables, so C and C' would be the same, written as: $(l_1 \vee l_2)$.

In either case, C is satisfiable if and only if C' is satisfiable. For the "if" direction, for the one literal case, $l = 1$ is any satisfying assignment of C', and at least one of $l_1, l_2 = 1$ in the two-literal case. And for the "only if" direction, $l = 1$ is a satisfying assignment for the one-literal case and at least one of $l_1, l_2 = 1$ in the two-literal case, proving (i) of Definition 1. Similar to (2), the formulas have a number of clauses which are linear in the number of gates and wires, as there is a direct one-to-one mapping between the use of clauses and logical operations with gates and wires, proving (ii) of Definition 1. Thus, it can be stated that ATMOST-2-CNF-SAT $\leq$ EXACTLY-2-CNF-SAT, so (3) is proven to be true.

To prove (4), given an instance of EXACTLY-2-CNF-SAT, i.e., a conjunction of clauses with exactly 2 literals each, construct a Boolean circuit C for the 2-CIRCUIT-SAT output, with each literal as a distinct input wire and each clause as a gate. Now, the EXACTLY-2-CNF-SAT formula is satisfiable if and only if the 2-CIRCUIT-SAT circuit is satisfiable, proving (i) of Definition 1. And the 2-CIRCUIT-SAT circuit has a number of gates and wires which are linear in the number of clauses, i.e., linear in the size of the formula, proving (ii) of Definition 1. Thus, it can be stated that EXACTLY-2-CNF-SAT $\leq$ 2-CIRCUIT-SAT, so (4) is proven true.

Since all four reductions in the procedure stated above are proven to be true, then by the property of transitivity, one can directly reduce from (1) straight to (4), namely k-CIRCUIT-SAT $\leq$ 2-CIRCUIT-SAT, completing the proof for this reduction.

Assignment 4

Question 4

To prove that if ISO ∈ **P**, then AUTO ∈ **P**, utilize a seemingly weaker approach of a reduction from Definition 1 of Lecture 9(c). In particular, if the ISO decision problem can be reduced to the AUTO decision problem, i.e., ISO ≤ AUTO, then the AUTO problem is at least as hard as the ISO problem, and since given ISO ∈ **P**, then it would hold that AUTO ∈ **P**. Since there is presumably a polynomial-time algorithm I for ISO, and the goal is to devise a polynomial-time algorithm A for AUTO, then a seemingly weaker notion of (ii) of Definition 1 is satisfied, namely that both I and A are polynomial-time computable. Now, take a seemingly weaker notion of (i) of Definition 1 for this problem: given an undirected graph $G = \langle V_G, E_G \rangle$, create a copy $H = \langle V_H, E_H \rangle$ of G, where $V_G \cap V_H = \emptyset$, and then alter both G and H in a way such that $I(G, H)$ correctly returns true or false. So, the goal here is to devise such an algorithm A that performs this method in polynomial-time, and this in turn would prove AUTO ∈ **P**.

The goal for the A algorithm is to properly modify G and H, for a polynomial number of times, and invoke the $I(G, H)$ algorithm for each pair of graphs. Furthermore, each modification to the graphs should also take polynomial-time. For the approach, suppose given the graph G and a copy of it H. Now, pick a pair of distinct vertices u, v and force a mapping of u onto v, immaterial of how the rest of the vertices are mapped. Now, extend this approach to every pair of distinct vertices u, v, to try and force a mapping of u in G onto v in H. Thus, a valid mapping exists if and only if G is automorphic. To force a mapping, “hang” something off of each of u and v that forces any isomorphic mapping to map u onto v only. More specifically, the A algorithm should construct a graph that forces such a mapping of u to v only.

To solve this problem, a polynomial-time algorithm A for AUTO was written in C++ to be able to test with various inputs for the graph G:

```cpp
bool I(vector<int>& G, vector<int>& H) {
    // Polynomial−time algorithm to determine isomorphism
}

void isAutomorphic(vector<int>& G, vector<int>& H, int pos,
bool& result) {
    if (pos == H.size() - 1) {
        if (I(G, H) && G != H) {
            result = true;
        }
        return;
    }
    if (result == true) {
        return;
    }
    for (int i = pos; i < G.size(); i++) {
        swap(H[i], H[pos]);
        isAutomorphic(G, H, pos + 1, result);
        swap(H[i], H[pos]);
    }
}

bool A(vector<int> G) {
    vector<int> H = nameChange(G);
    bool result = false;
    isAutomorphic(G, H, 0, result);
    if (result == true) {
        return true;
    }
    return false;
}
```

The program consists of three functions: the algorithm I to determine isomorphism of two input graphs, a helper function $isAutomorphic$ to carry out the automorphism evaluation of a graph, and the polynomial-time algorithm A to determine automorphism of the input graph. To start, the algorithm A takes in as input an undirected graph, call it $G = \langle V_G, E_G \rangle$, and then makes a copy of it, call it $H = \langle V_H, E_H \rangle$, where it is guaranteed that $V_G \cap V_H = \emptyset$. The utility function $nameChange$ makes a copy of the input graph and linearly changes the vertex data such that none of them are the same, namely $V_G \cap V_H = \emptyset$. Then, the $isAutomorphic$ method is invoked passing in both graphs, as well as a starting value of 0 for a position tracker and a Boolean result variable to store the decision problem solution.

The $isAutomorphic$ method is the main function to perform the automorphism evaluation and is implemented as a recursive solution. The idea is that it continuously modifies the graph H and then checks for isomorphism each time. For the base case, if the modification has reached the last vertex of the graph H, then do the following: if the graphs G and H in their current state are isomorphic, i.e., $I(G, H)$, and they are not identical, i.e., not the identity mapping, then set the solution to true, meaning the graph G is indeed automorphic. At the end of the base case, simply return to step out of that recursive call stack, since it is done with evaluating that instance of H.

The other end condition is if it has indeed determined a solution, then short circuit the algorithm and terminate there with a return.

If it is not at the last vertex of H, perform the recursive step. Iterate from the current position as stored in the position tracker up to the size of the graphs and do the following: perform a modification to the graph H using the index and position trackers to generate new versions of H for evaluation. For each new version, recursively invoke $isAutomorphic$ with an increment to the position counter to continue the evaluation. Finally, once a return happens as a result of the base case, undo the modification to evaluate more versions of H. This ensures the algorithm appropriately considers all the viable versions of H that it could possibly take to achieve correctness. If the algorithm goes through all the possible versions without converging to a solution, then this means the graph G is not automorphic.

The proposed algorithm above for A takes as input the graph G, creates a copy H from G where $V_G \cap V_H = \emptyset$, and then methodically alters the graph such that $I(G, H)$ correctly returns true or false. Within the algorithm, the I function is invoked multiple times, namely every time a modification is made to H to evaluate new versions. Furthermore, in the scenario that the outcome is the identity mapping, i.e., guaranteed to return true or the trivial result, the algorithm handles this case and effectively ignores it. Instead, it seeks to attain a unique and meaningful result for the automorphism.

Analyzing the time efficiency of this algorithm, the $isAutomorphic$ helper function performs the majority of the work for it. In this method, it iterates through each of the vertices in H, and for each vertex, it again iterates through each of the vertices to perform the modifications. For each modification, it calls the I method for ISO, which itself is a polynomial-time algorithm. The modifications themselves are constant time operations, however iteration through the graph H is quadratic time in the worst case. The constant time operations can be omitted; therefore, the algorithm is upper bounded by quadratic time efficiency.

As a result, this leads to A being a polynomial-time algorithm for AUTO. With that being said, both I and A are polynomial-time algorithms, meaning they are equivalent in terms of "hardness" or computational complexity. And since A correctly determines automorphism, while utilizing I for determining isomorphism, this leads to the desired conclusion: I is a polynomial-time algorithm and ISO ∈ **P**, and since A is as computationally complex as I, then AUTO ∈ **P**.

Assignment 4

Question 5

(a) **Proof that VERTEX-COVER-BOUNDS is NP-Hard**

To show that VERTEX-COVER-BOUNDS is **NP**-hard, utilize Definition 1 from Lecture 9(c) to reduce from some problem that is already known to be **NP**-hard to this problem. Naturally, an analogous problem that can be used for this reduction and is known to be **NP**-hard is VERTEX-COVER. So, if it can be shown that VERTEX-COVER ≤ VERTEX-COVER-BOUNDS, and since VERTEX-COVER is **NP**-hard, then this will prove that VERTEX-COVER-BOUNDS is also **NP**-hard.

The VERTEX-COVER optimization problem is the problem of finding the minimum-sized vertex cover, which is the smallest set of vertices of a graph that contains at least one endpoint of each edge of the graph. As for VERTEX-COVER-BOUNDS, this decision problem involves finding two properties: (1) existence of a vertex cover of G where its size is upper-bounded by k_1, and (2) evaluation of every vertex cover of G where their sizes are lower-bounded by k_2. There is a natural connection between these two problems, namely that the algorithm for VERTEX-COVER is directly used in the VERTEX-COVER-BOUNDS problem.

For (1), the algorithm for VERTEX-COVER is directly used here: to check for existence of a vertex cover of G where its size is upper-bounded by k_1, simply compute the optimal (minimum-sized) vertex cover of G. This is the correct way to check because this part is just looking for any such vertex cover that satisfies this property, and the minimum-sized vertex cover is the appropriate way to test this. Larger sized vertex covers may be larger than k_1, which does not satisfy the property, but that does not mean that this property cannot be satisfied, as smaller vertex covers may suffice. Thus, for (1), any instance of the algorithm for VERTEX-COVER returns a vertex cover that satisfies this property if and only if the VERTEX-COVER-BOUNDS problem also satisfies this property.

For (2), to check if for every vertex cover, that their sizes are lower bounded by k_2, simply test the largest viable vertex cover of G and check if its size is greater than or equal to k_2. The largest vertex cover of G is the set of all vertices of G, since this set would contain at least one endpoint of each edge of the graph. This is the correct way to check because if the size of this vertex cover is not greater than or equal to k_2, then none of the other smaller vertex covers will be greater than or equal to either. Similarly, if the size of this vertex cover is less than k_2, then immediately return false since not every vertex cover is of size $\geq k_2$. Thus, for (2), trivially take the largest vertex cover, which is the set of all vertices of G, and this property is satisfied if and only if this set satisfies it. There is no need to check any other vertex cover for this property.

Since both properties of VERTEX-COVER-BOUNDS above can be satisfied if and only if VERTEX-COVER itself is utilized within the problem, then (i) of Definition 1 is satisfied. Now, analyzing the time efficiency of VERTEX-COVER-BOUNDS, it can again be split into two parts. For checking the condition involving k_1, the time efficiency of this part of the algorithm is correlated to the time efficiency of the algorithm for VERTEX-COVER since it is directly used here. The check for existence at the end for a vertex cover of size $\leq k_1$ can be done in constant time, therefore the VERTEX-COVER time efficiency is dominant in this case.

For checking the condition involving k_2, the algorithm has to traverse the entire graph to count up the number of vertices, which can be done in linear time, namely $\Theta(|V| + |E|)$. Once again, since only one vertex cover has to be evaluated, namely the set of all vertices in G, then the check for size $\geq k_2$ can be done in constant time.

Comparing the time efficiencies of each part, the one for VERTEX-COVER clearly dominates, since the second part is only linear time, therefore the VERTEX-COVER-BOUNDS problem is upper-bounded by the time efficiency of VERTEX-COVER. Thus, they are equivalent in terms of hardness or computational complexity, so (ii) of Definition 1 is satisfied. Therefore, it can now be claimed that VERTEX-COVER reduces to VERTEX-COVER-BOUNDS, or VERTEX-COVER $\leq$ VERTEX-COVER-BOUNDS, and finally, since VERTEX-COVER is already known to be **NP**-hard, then VERTEX-COVER-BOUNDS is also **NP**-hard.

(b) **Computational Complexity of VERTEX-COVER-BOUNDS**

The VERTEX-COVER-BOUNDS problem is in **NP** because (i) it is a decision problem, and (ii) for any instance of this problem where the result is true, there exists a proof or deterministic polynomial-time algorithm to verify the correctness of the result. Moreover, this problem is directly related to the VERTEX-COVER problem which is already proven to be **NP**-complete. This problem is upper-bounded by the time efficiency of VERTEX-COVER, meaning it is no more computationally complex than it. The only additional step it has to take is a full traversal through the graph which only takes linear time, so it is dominated by VERTEX-COVER. At the same time, it cannot be more efficient or less "hard" than VERTEX-COVER, since it has to at least determine the minimum vertex cover, so it cannot be in the class **P**, thus it must be in **NP**.