Problem Statement 1

Given two graphs G = (VG, EG) and H = (VH , EH), G is said to be isomorphic to H if there is a bijection (namely, one-to-one mapping) between the two sets of vertices VG and VH: f : VG → VH such that for each pair of vertices u, v VG, they are adjacent in G if and only if the two corresponding vertices f(u), f(v) VH are adjacent in H (namely, (u, v) EG if and only if (f(u), f(v)) EH). Clearly, if G is isomorphic to H, then H is isomorphic to G, and vice versa. In other words, if G and H are isomorphic, then they are essentially the same graph, except that they label their vertices differently. The Subgraph-Isomorphism Problem takes two undirected graphs G1 and G2, and asks whether G1 is isomorphic to a subgraph of G2. Show that the subgraph-isomorphism problem is NP-complete.

Main Idea 2

The Subgraph-Isomorphism Problem takes two undirected graphs G1 and G2, and asks whether G1 is isomorphic to a subgraph of G2.

To begin solving this problem, first we need to prove that this situation is at least NP. To prove this, we need to check a few things: check that G1 and subgraph G2 have the same number of vertices, and check that G1 and G2 have the same edges connecting vertices. Since you can easily and efficiently check these properties in polynomial time, then we know that this problem is at least NP.

Next, we need to prove that this situation is also NP-hard. We can prove this with the Clique Problem. A clique is a subset of graph G where each pair of vertices is connected by an edge. We want to determine if a graph contains a clique of size n. If G1 is a complete graph with n vertices, and G2 is a graph that we want to find a clique of size n in. If G2 contains this specified clique of size n, then that means that G2 will be isomorphic to G1, which is the Subgraph Isomorphic Problem. However, if we started from the assumption that the Subgraph Isomorphic Problem is true, then that means G2 would need to form a complete graph with n vertices, thus meaning that it also has a clique of size n. Thus, since the Clique Problem is known to be NP-complete and can be reduced to the Subgraph Isomorphism Problem, then this problem is also NP-hard. Since the Subgraph Isomorphism Problem is both at least NP and NP-hard, then it is also NP-complete.

Pseudocode 1

Algorithm 1 Question 1 - Subgraph Isomorphic Problem

1: **function** SubgraphIsomorphicProblem(void)
2: Initialize global variables:

- G1 = Graph 1 with vertices V1 and edges E1
- G2 = Graph 2 with vertices V2 and edges E2
- VertexDict = initialize to empty dictionary
- For loop: For i in 1 to length of V1
  - ValidMapBool = False
  - For loop: for j in V2
    - if j is not in VertexDict
      - Add to dictionary
      - ValidMapAdjacency = True
      - For loop: For each pair in VertexDict (a, b):
      - - If a != b, check that adjacency is consistent between the two graphs. The adjacent vertices must match for both graphs. If they do not, make ValidMapAdjacency = False
      - If ValidMapAdjacency = True
      - - ValidMapBool = True. This means mapping is valid
      - - Break out of inner loop
      - If ValidMapAdjacency = False
      - - Remove i mapping from VertexDict
    - If ValidMapBool = False
      - Return("False"). This means the graphs were not isomorphic
    - Else
      - Return("True")

3: **end function**

Proof of Correctness 1

This algorithm should be correct because it will explore all possible mappings between the vertices of G1 and G2. This means that any potential subgraph isomorphism will be checked. The algorithm will begin checking the vertices between G1 and G2, continuing until it reaches a pair that does not meet the desired adjacency. When this occurs, the algorithm will remove that current pair mapping and try another possibility. This will continue until either a

valid subgraph isomorph is found or all possibilities have been checked. If the adjacency conditions are satisfied for all vertices and edges, the algorithm will correctly identifies that G1 is isomorphic to a subgraph of G2. However, if it fails the adjacency condition for all vertices, the algorithm will return that no subgraph isomorphism exists.

Time Complexity 1

For overall time complexity, we need to look at the time complexity for each loop.

The outer loop iterating over G1 will iterate a maximum of V1 times, meaning it has a time complexity of O(V1). The inner loop iterating over V2 will iterate a maximum of V2 times, so it has a time complexity of O(V2). Checking valid adjacency will conduct up to i*i checks due to the pairs, so it has a time complexity of (V1 * V1). This means that the overall time complexity for this algorithm will be found by multiplying everything together. The overall time complexity is O(V1 * V2 * V1 * V1).

Problem Statement 2

An Independent Set of a graph G = (V, E) is a subset V V of vertices such that each edge in E is incident on at most one vertex in V. The Independent-Set Problem is to find a maximum-size independent set in G. Formulate a related decision problem for the independent-set problem, and prove that it is NP-complete.

Main Idea 2

In this scenario, V' is an independent set which is a subset of graph G. The Independent-Set Problem is to find a maximum-size independent set in G. An independent set is a set of vertices, where no two are adjacent to each other.

To begin solving this problem, its important to frame this based on the main decision: is there a subset V' with n number of vertices, where no two vertices are adjacent?

To prove that this approach is at least NP, we can start with a proposed solution containing n number of vertices. To determine if this solution is an independent set, there are two main things that need to be checked: the size of the set and that there are no edges between any set of vertices. Since you can easily and efficiently check these properties in polynomial time, then we know that this problem is at least NP.

Next, we need to check that the problem is NP-hard, where it is at least as hard as any NP-problem. We can approach this with the Hamiltonian Cycle Problem, where we're looking for a cycle that goes through every node and vertex exactly once. Suppose we use our given graph G, and construct a new graph, G', where each vertex in G is replaced with a pair of vertices; in these pair of vertices, one represents the vertex itself and the other represents a new vertex. These new vertices are connected by edges within the new graph so that the

independent set selected in G corresponds to vertices that form a cycle in G. The size of this independent set can then be labeled as k, where k = V so that V is a set of vertices from G. If G contains a Hamiltonian cycle, then in G it should be possible to select one vertex from each pair corresponding to the vertices in the Hamiltonian cycle, forming an independent set of size V. Conversely, if we can find an independent set of size V in G, then the corresponding set of vertices in G should also form a Hamiltonian cycle. The independence of the set ensures that no two selected vertices are adjacent. Since we can validate the existence of an independent set of size k in G', then a Hamiltonian cycle should exist. Since the Hamiltonian Cycle Problem can be reduced to the Independent Set Problem in polynomial time, then the Independent Set Problem can be NP-hard. Thus, since the Independent Set Problem is both NP and NP-hard, the problem is also NP-complete.

Pseudocode 2

Algorithm 2 Question 2 - Independent Set Problem

1: **function** IndependentSetProblem(void)
2: Initialize global variables:
- G = graph with V vertices and E edges
- IndependentSet = initialize to empty set
- UnvisitedVertices = initlialize to equal to all possible vertices in V
- While loop: while UnvisitedVertices is not empty:
  - currentVertex = select a vertex from UnvisitedVertices
  - Add currentVertex to IndependentSet
  - For all neighboring adjacent vertices:
    - Remove neighbor from Independent Set
  - Remove currentVertex from IndependentSet

3: Once you exit the while loop, you've reached the end of the algorithm. Return(IndependentSet)
4: **end function**

Proof of Correctness 2

To get a completely correct answer, one would like need to use brute force to check every vertex. However, this can get computationally expensive very quickly.

This algorithm works as an efficient approximation of what the maximum independent set should be. This algorithm considers the most important part of the solution to just ensure that the selected set is long and independent, but not necessarily the longest. As the algorithm chooses a vertex from the UnvisitedVertices, it removes all adjacent vertices from

being checked, which ensures that the final set is independent. No two adjacent vertices can be selected or examined.

Utilizing a greedy algorithm for this solution means that expected optimal choices for each vertex are being made, even if they may not be the optimal solution for the actual max independent set. We assume that each of the vertices that get checked, should be part of the final independent set rather than any of its adjacent neighboring vertices.

Time Complexity 2

For this algorithm, the overall time complexity is primarily based on the while loop. The while loop will iterate at most V times, if it has to look at each of the individual vertices. Additionally, the for loop used to remove all neighboring adjacent vertices will iterate at most E times. Thus, the overall time complexity of this algorithm is O(V + E).