Question 1

(a) **Letter A** can not be inferred because it could be possible that X is only in NP, Therefore the statement would be false.

(b) **Letter B** can not be inferred because it could be possible that Y is only in NP-Hard, therefore proving the original conjecture to be false.

(c) **Letter C** can not be inferred because it is very possible for X to be in the P realm.

(d) **Letter D** can be inferred due the fact that Y can be in NP and NP-complete.

(e) **Letter E** Can not be inferred since X and Y could be NP-complete, if we take X which is NP-complete and reduce it to Y, then Y, it will also be NP-complete.

(f) **Letter F** can not be inferred since is can said that Y can be harder than x, Therefore proving the original conjecture to be false.

(g) **Letter G** can be inferred since we know that Y is no harder than X.

Question 2

To show that the HAM-PATH is NP-complete we must have a couple of ideas/concepts in mind: First, we need to shows that the HAM-PATH that exists in a graph is in NP; we can say that we are given a solution to the HAM-PATH; having the solution we are able to verify it in polynomial time (NP). To verify in polynomial time, all we need to do is take the answer, the path through the graph, and verify that all nodes are traversed and there are no repeated nodes in the traversal; while also checking that the starting node is u and the ending node is v. [1][2][3]

Second and lastly, we need to keep in mind that we can reduce a HAM-CYCLE to a HAM-PATH, please refer to figure 1 for the reduction illustration. For this example we can say that graph G contains the Hamiltonian cycle and next to G we have G' which contains the HAM-PATH, to get G' we simply choose an arbitrary vertex u (in this case A) in G and adding a copy, u' (A'), of it together with all its edges. Then add vertices V and V' to the graph and connect V with A and V' with A'. In figure 1 the suppose that a HAM-CYCLE exists in G then the cycle will be as follows: $A \to F \to C \to D \to B \to E \to A$ notice how it starts and ends on the same vertex and does not repeat node traversals. Then would get the HAM-PATH from G': if we take V to (A) and follow the path we got from G and end at A' to V'; following these steps we get Hamiltonian path to be: $V \to A \to F \to C \to D \to B \to E \to A' \to V'$.

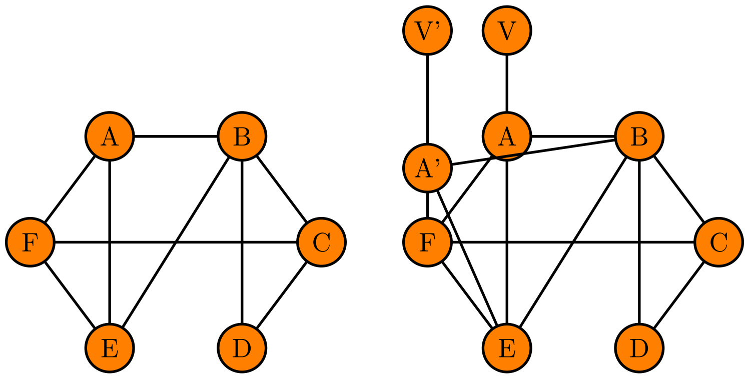

Figure 1: : A graph G (left) and the Hamiltonian path reduced graph G' (right).

Conversely, now suppose G' contains a HAM-PATH. In this case, the path must have endpoints in V and V'. This path that is followed in G' can be transformed to a HAM-Cycle in G. To transform the path to a cycle we would simply disregard V and V', but the path must still have endpoints in A and A' and if we remove A' we get a cycle in G; this means we will close the path back to A instead of A'. In this example I have shown that G has a HAM-CYCLE if and only if G' contains a HAM-PATH. Therefore, concluding and proving that HAM-PATH is in NP-Complete.

Question 3

To show that Long Path is in NP-complete we must first have a few couple of concepts/ideas in mind: First, we need to look see whether Long Path is in NP, this is very simple to verify; suppose we are given the solution to Long Path, to verify the path in polynomial time we all we need to do would be to verify that the edges are in G and that path is at least k where no nodes are repeated in the traversal. From this verification process we show that LONG-PATH is in NP.[1][2][3]

Next, we need to show that HAM-PATH is NP-complete; which we prove in question 2 using figure 1. Since we know that HAM-PATH reduces to LONG-PATH.The reduction follows directly from HAM-PATH. Given an instance of HAM-PATH on a graph $G = (V, E)$. We create an instance of the longest path problem G' , k as follows. We use exactly the same graph, such that $G' = G$ and we set $k = |V| - 1$. Then there exists a simple path of length k in G' if and only if G' contains a HAM-PATH; this shows that HAM-PATH $\propto$ LONG-PATH.

Using the information from points 1 and 2, we can therefore conclude and successfully show that LONG-PATH is NP-complete.

Question 4

(a) Essentially this question is asking for whether a graph is bipartite or not, hence the two color distinction, an effective algorithm to solve this would be a "graph bipartiteness finder" program which is essentially a modified version of the breadth first search, using an adjacency list, to achieve the most efficient runtime complexity. In the traversal of the trees assign all nodes whose distance is even to be blue (or desired color) and all nodes whose distance is odd to be red (or desired color). Then check each edge to verify that it goes between blue and red nodes. The following 4 chunks of pseudo-code will show how the bipartite check process is done, I chose to split the code up since its makes the reading a little bit easier. [1][2][4]

```
(class data type ) Graph {
    V; // vertex number
    <list structure> adj_list; // adjacency list
public:
    list_accessor (); // list accessor
    Graph(int V); // constructor
    ¬Graph(); // descructor
    add_edge(int v, int w); // add edge to graph function
    print_list (); // prints list , used for testing purposes
    is_bipartite_main (int **Graph, int team_array[]); // bipartite main
    is_bipartite_helper(int **Graph, int source, int team_array[]);// bipartite helper
};
```

In the pseudo-code above I use a class that mimics a graph, holds vertex information, and has and built in adjacency list structure to maximize efficiency. There are 5 trivial functions that are in the class that I will not provide pseudo-code for: that would be list_accessor function which is used to access the the adjacency list outside of the class object. The constructors and destructor for the class. In the constructor I initial the list to point to NULL since it is just initialization. The destructor I implemented simply frees all of the dynamically allocated memory from the heap. Next, we have the print_list function which is used to print out the adjacency list; this function is mainly for debugging and looking at list values. Lastly, we have the add_edge function which essentially adds a new vertex to the adjacency list. It is also worth noting that since I am using an adjacency list, the complexity will be $V + E$.

```
Bipartite_main_function (Graph, team)
  for i = 0 to V
    team[i] = -1 // -1 initialization

  // This code is to handle disconnected graph
  for i = 0 to V
    if (team[i] == -1) // if not teamed, run
      if (is_bipartite_helper(Graph, i, team_array) == false) // bipartite_helper run
        return false

  return true
```

The code above is the pseudo-code is the main function that calls the bipartite_helper to check to see of the graph that is passed in is bipartite or not, doing this function also handles any disconnects in the graphs, using lines 6-8, the for loop will ignore the disconnects in the graph so ensure that the results do not get skewed.

```
bipartite_helper (Graph, source, team)

  team[source] = 1; // selects the first passed in node to be 1

  using a queue structure (of integers) q
    q.push(source);

    // while the queue is not empty run the bfs, alogithm
    while q not empty
        head = q.front();
        q.pop();

        if (the there are the same designation for two nodes)
           return false

        // this will used to run thru the color array to check the
        // the teams, whether a node is 0 or 1, if they are both then
        // the return will be false.
        for i = 0 to V

            // if the node is node assigned yet, then run.
            // An edge from head to v exists and destination v is not colored
            if current_node (head) and corresponding team array equal -1
                assign alternate color
                q.push(v)

            if the nodes are the same color in that edge
                return false // return no destination available


  return true
```

The pseudo-code above show the function of the assigning/designating function of the code, in this function the odd (blue) and even (red) nodes will get separated, evens will be 1s, and odds will be 0s. 0's will signify the reds, while the 1's will signify the blues. To do this the algorithm takes a "color array" (binary, 1 equals blues, 0 reds) and sorted them, after the sorting is completed the filled color array will then be used to determine which nodes are blue and which nodes are red. Essentially this code combined with bipartite_main is just a heavily modified version of the original BFS algorithm found in the class textbook (CLRS).

Now for the time complexity of the "graph bipartiteness finder" program since we are using an adjacency list the time complexity will be **O(V+E)** where V is a number of vertices in the graph, and the E corresponds to the number of edges in the graph.

(b) To prove that 4-coloring is NP-complete we must utilize the fact that 3-coloring is NP-complete. We can prove that the 4-coloring problem is NP complete by a polynomial-time reduction from the 3-coloring problem, we will also have to assume that we have a function to solve the 4-coloring problem using polynomial time (NP). Now, suppose we are given an instance of the 3-Coloring problem: a graph $G = (V, E)$. We then create a new graph G' from G by adding a new vertex v' and adding an edge v, v' for every $v \in V$. We then provide the graph G' to our function for solving the 4-coloring problem. The output of that function is a single bit telling us whether or not G' has a 4-Coloring. We return that same bit as the solution of the 3-Coloring problem. The reduction is made possible by the following claim: G can be 3-Colored if and only if G' can be 4-colored.

We know that the claim used above is true when take a look at the following example: suppose G can be colored with 3 colors. Then G' can be colored with 4 colors: just assign the vertex v' the color "4". Conversely, suppose that G' can be 4-colored. Then v' uses a color that is not used by any other vertex in V . So the remaining colors are a 3-coloring of G, in this case the color numbers should make the problem. i.e if it is a 3-color then color 1, 2, and 3 should exist.

Using paragraphs 1 and 2 it is therefore concluded that the 4-coloring problem is NP-complete.

References

[1] G. for Geeks, "Np-completeness | set 1 (introduction)." https://www.geeksforgeeks.org/np-completeness-set-1/, 2013.

[2] S. Overflow, "What is an np-complete in computer science?." https://stackoverflow.com/questions/210829/what-is-an-np-complete-in-computer-science, 2008.

[3] G. for Geeks, "Backtracking | set 6 (hamiltonian cycle)." https://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/, 2012.

[4] Wikipedia, "Graph coloring." https://en.wikipedia.org/wiki/Graph_coloring, 2018.