Notes on my pseudocode: I’m using Java-like syntax where everything is an object, and I can add properties to any object without declaring them, e.g. if v is a vertex I can set v.mapped = true and that vertex object will carry that boolean with it. Everything is also case-sensitive. The syntax G.E(v1,v2) where G is a graph and E is its set of edges is true if the edge v1-v2 is in G.E, false otherwise.

Question one

Give a poly-time verification algorithm for GRAPH-ISO and justify its running time.

```
// Input: Graphs G1 and G2 (with vertices G1.V and G2.V and edges
// G1.E and G2.E) and bijective mapping function f to verify
// Output: True if function f satisfies the Graph-ISO problem
// specification, false otherwise
Verify-Graph-ISO(G1, G2, f)
  if (G1.V.size != G2.V.size)
    return false
  for each v in G2.V
    v.mapped = false
  for each v1 in G1.V
    v2 = f(v1)
    if (v2 is not in G2.V)
      return false
    if (v2.mapped == true)
      return false
    v2.mapped = true
    for each v1neighbor in G1.V
      v2neighbor = f(v1neighbor)
      if G1.E(v1,v1neighbor) and !G2.E(v2,v2neighbor)
        return false
      if !G1.E(v1,v1neighbor) and G2.E(v2,v2neighbor)
        return false
  return true
```

The algorithm first checks that G1 and G2 have the same number of vertices. Then for each vertex in G1, it checks that it maps to an unused vertex in G2, and then checks that any vertex the one in G1 is adjacent to, its mapped partner in G2 is also adjacent to (with respect to mapped adjacent vertices in G2).
Let $v = G1.V.size = G2.V.size$:

$$T(n) = c_1 + \left( \sum_{i=1}^{v} c_2 \right) + \left( \sum_{i=1}^{v} c_3 + \left( \sum_{i=1}^{v} c_4 \right) \right)$$

$$= c_1 + c_2v + c_3v(c_4v)$$

$$= c_1 + c_2v + c_3c_4v^2$$

$$= O(v^2)$$

$O(v^2)$ is of the form $m^n$ where $n$ is constant (2), so this verification runs in polynomial time.
Question two

Give a poly-time verification algorithm for HAM-PATH and justify its running time.

// Input: Graph G and vertices u and v which are the start and end
// of the Hamiltonian path. path is the list of vertices to verify.
// Output: True if the given path is a valid Hamiltonian path from u to v,
// false otherwise.
Verify-Ham-Path(G, u, v, path)
    if (path.size != G.V.size)
        return false
    if (path.firstVertex != u)
        return false
    if (path.lastVertex != v)
        return false
    for each v in G
        v.visited = false
    for each v in path except last
        if (v.visited == true)
            return false
        v.visited = true
        vnext = path.vertexAfter(v)
        if (v is not in G.V)
            return false
        if !(G.E(v, vnext))
            return false
    return true

Verify-Ham-Path first checks that the path is of the correct length to cover the entire graph, and
that it is starts at u and ends at v. Then it follows the path checking that each vertex is connected
to the next in G and that no vertices are repeated.

Let v = G.V.size, p = path.size = v (worst case):

\[
T(n) = c_1 + \left( \sum_{i=1}^{v} c_2 \right) + \left( \sum_{i=1}^{p} c_3 \right) \\
= c_1 + \left( \sum_{i=1}^{v} c_2 \right) + \left( \sum_{i=1}^{v} c_3 \right) \\
= c_1 + c_2v + c_3v \\
= c_1 + c_2c_3v^2 \\
= O(v^2)
\]

\(O(v^2)\) is of the form \(m^n\) where \(n\) is constant (2), so this verification runs in polynomial time.
Question three

Prove LONGEST-CYCLE is NP-complete. Justify the verification algorithm’s running time, why the reduction works, and why the reduction is poly-time.

// Input: Graph g, bound k, and list of vertices cycle.
// Output: True if cycle is a simple cycle of length k or greater in graph G.
Verify-Longest-Cycle(G, k, cycle)
    if (cycle.length < k)
        return false
    if (cycle.first != cycle.last)
        return false
    for each v in G.V
        v.used = false
    for each v in cycle except last
        if (v.used == true)
            return false
        v.used = true
        vnext = cycle.vertexAfter(v)
        if !(G.E(v, vnext))
            return false
    return true

First the cycle is checked to make sure it is of the correct length and starts and ends at the same vertex. Then each vertex is checked in the cycle to make sure they’re connected, and, being a simple cycle, only the first and last vertices are repeated.

Let v = G.V.size, p = path.size = v+1 (worst case):

\[
T(n) = c_1 + \left( \sum_{i=1}^{v} c_2 \right) + \left( \sum_{i=1}^{v-1} c_3 \right)
\]

\[
= c_1 + \left( \sum_{i=1}^{v} c_2 \right) + \left( \sum_{i=1}^{v} c_3 \right)
\]

\[
= c_1 + c_2v + c_3v
\]

\[
= c_1 + c_2c_3v^2
\]

\[
= O(v^2)
\]

\(O(v^2)\) is of the form \(m^n\) where \(n\) is constant (2), so this verification runs in polynomial time.
Reduction algorithm:

// Input: Graph G
// Output: True if a simple cycle that visits each vertex
// in G exists, false otherwise
Ham-Cycle(G)
    // first and last vertex are repeated, so add 1 to size
    if (Longest-Cycle(G,G.V.size+1)==true)
        return true
    return false

This reduction is constant time as it contains no loops and only a single call to Longest-Cycle.

It asks Longest-Cycle if a cycle of length V.size+1 exists in G. It actually asks for that length or longer, but a longer cycle is impossible because you would have to repeat more than one vertex. A simple cycle only repeats one vertex, the first and last, and every other one is unique. Therefore it covers every vertex in G.

In other words, a simple cycle of length V.size+1 is a Hamiltonian cycle.