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**

*Prove SUB-ISO is NP-complete.*

```
// Input: Graphs G1 and G2, and function f which maps vertices
// in G1 to G2, or null if there is no mapping
// Output: True if f maps some or all vertices in G1 to a subgraph
// of G2 that is isomorphic to the corresponding G1 vertices.
// False otherwise.
Verify-SUB-ISO(G1, G2, f)
  for each v in G2.V
    v.used = false
  for each v in G1.V
    vmapped = f(v)
    if (vmapped is not in G2.V)
      return false
    if (vmapped.used == true)
      return false
    vmapped.used = true
    for each vadjacent in G1.V
      if (G1.E(v,vadjacent)==true)
        if (G2.E(f(v),f(vadjacent))!=true)
          return false
  return true
```

The first loop runs over \(v\) vertices. The second loop is a double loop, both going over \(v\) vertices totalling \(v^2\). Everything else in the double loop is constant time. Therefore the total running time is \(\Theta(v^2)\).
Reduction:

Clique(G, k)
    if (k > |G.V|)
        return false
    C = new graph with k vertices
    for each v in C
        for each av in C // adjacent vertices
            if (v != av)
                set vertex v adjacent to av in C
        if (SUB-ISO(C, G) == true)
            return true
    return false

The reduction constructs a new graph C that is a full Clique of size k. The SUB-ISO algorithm will search for a subgraph in G that is fully connected, which will be a clique. If such a subgraph exists, that is a clique of size k and both return true. If there is no clique in G, SUB-ISO will fail because no fully connected subgraph will exist in it.

The reduction creates a graph C in $k$ time, and populates it with a double loop over $k$ vertices totalling $k^2$. The inside is just a conditional and an assignment, both constant time operations. Therefore the total reduction time is $\Theta(k^2)$.
**Question two**

a) Execute GreedyColoring on $V=\{1,2,3,4,5\}$ and $E=\{(1,2),(2,3),(3,4),(4,5)\}$. Does the algorithm find the optimal coloring? If not, give it.

Step 1:

$v=1$

adjacent colors=none

action: color vertex 1 with color 1

coloring: \{1,-,-,-,-\}

Step 2:

$v=2$

adjacent colors=1 (vertex 1)

action: color vertex 2 with color 2

coloring: \{1,2,-,-,-\}

Step 3:

$v=3$

adjacent colors=2 (vertex 2)

action: color vertex 3 with color 1

coloring: \{1,2,1,-,-\}

Step 4:

$v=4$

adjacent colors=1 (vertex 3)

action: color vertex 4 with color 2

coloring: \{1,2,1,2,-\}

Step 5:

$v=5$

adjacent colors=2 (vertex 4)

action: color vertex 5 with color 1

coloring: \{1,2,1,2,1\}

Yes, \{1,2,1,2,1\} is the optimal coloring.
b) For \( V=\{1,2,3,4,5,6\} \) and \( E=\{(1,4),(1,6),(2,3),(2,5),(3,6),(4,5)\} \), how many colors are used by the GreedyColoring algorithm? How many for the optimal coloring?

Step 1:
\( v=1 \)
adjacent colors=none
action: color vertex 1 with color 1
coloring: \( \{1,-,-,-,-,-\} \)

Step 2:
\( v=2 \)
adjacent colors=none
action: color vertex 2 with color 1
coloring: \( \{1,1,-,-,-,-\} \)

Step 3:
\( v=3 \)
adjacent colors=1 (vertex 2)
action: color vertex 3 with color 2
coloring: \( \{1,1,2,-,-,-\} \)

Step 4:
\( v=4 \)
adjacent colors=1 (vertex 1)
action: color vertex 4 with color 2
coloring: \( \{1,1,2,2,-,-\} \)

Step 5:
\( v=5 \)
adjacent colors=1 (vertex 2) and 2 (vertex 4)
action: color vertex 5 with color 3
coloring: \( \{1,1,2,2,3,-\} \)

Step 6:
\( v=6 \)
adjacent colors=1 (vertex 1) and 2 (vertex 3)
action: color vertex 6 with color 3
coloring: \( \{1,1,2,2,3,3\} \)

The greedy algorithm uses three colors, but the optimal only needs two: \( \{1,2,1,2,1,2\} \).
c) What is the worst-case time complexity of GreedyColoring? Justify your answer.

The for each statement loops \( v \) times, and inside that loop potentially each vertex needs to be checked if it’s adjacent, which takes a total of \( c \cdot v^2 \). The first statement can be constant time and the second is \( c \cdot v \). So \( T(n) = O(v^2) \).
Question three

*Explain in detail why the dynamic programming solution for SubsetSum is not in P.*

The input consists of $n$ weights and a capacity $C$. The weights are all equal to or less than $C$, and a reasonable encoding of the weights would be binary, which uses $\lg n$ bits. The input size is therefore $(\lg C)(n + 1)$ in the worst case, since $C$ is also encoded using $\lg n$ bits.

The given running time is $nC$, which consists of two independent variables. $n$ is linearly related to $(\lg C)(n + 1)$, since $n$ itself appears in the expression. $C$ is exponentially related, because $\lg C$ needs to be raised to the power of 2 to equal $C$ ($2^{\lg C} = C$). The exponential relation forces the overall running time to be exponential, and therefore not polynomial.