```
=============================================================================
CSC 373                Tutorial Solutions for Week  9            Winter 2019
=============================================================================


 1. >>  I present the solutions in a question-and-answer format: please   <<
    >>  ask each question and get students to think about them before     <<
    >>  you go over the answers with them. This is one instance where it  <<
    >>  would make more sense to present the material at the front of     <<
    >>  the room, lecture style.                                          <<

    Q:  What are the two main steps to show D is NP-complete?
    A:   1. Show D in NP.
         2. Show D is NP-hard.

    Q:  How do we show D in NP?
    A:  Describe a polytime verifier for D -- an algorithm V(x,c) with the
        following properties:
          * V(x,c) runs in polytime (as a function of size(x));
          * for all x in D, there exists c such that V(x,c) = True;
          * for all x not in D, for all c, V(x,c) = False
            [equivalently, for all x in the set of inputs for D,
            if V(x,c) = True for some c, then x in D].

    Q:  How do we show D is NP-hard?
    A:  Show E -p> D for some decision problem E known to be NP-hard.

    Q:  How do we show E -p> D?
    A:  Describe a transformation algorithm T with the following properties:
          * T takes inputs x for E and generates inputs T(x) for D;
          * T runs in polytime for all inputs x (whether in E or not);
          * for all x in E, T(x) in D;
          * for all x not in E, T(x) not in D
            [equivalently, for all T(x) in D, x in E].

    Q:  What are some of the important points that are easy to forget (or to
        get mixed up about)?
    A:    * T only gets input x -- *not* certificate c. For example, if
            doing a reduction from 3SAT to D, T gets a formula in 3CNF but
            *not* how to set truth values to make the formula True (this may
            not even be possible). In general, keep in mind that T must work
            for every input x, whether the answer is Yes or No. For example,
            if doing a reduction from 3SAT to D, T must transform every
            formula into an input for D -- whether the formula can be
            satisfied or not.
          * The argument that T works must show x in E iff T(x) in D -- you
            *cannot* do this by describing two separate transformations.
            For example, you cannot describe how to transform a satisfiable
            formula into an input in D, and separately how to transform an
            input in D into a satisfiable formula. You must describe how to
            transform *any* formula into an input for D, and then argue that
            satisfiable formulas become yes-instances for D and
            unsatisfiable formulas become no-instances for D.
          * The argument that x in E iff T(x) in D is typically done in two
            steps: show x in E implies T(x) in D, then show T(x) in D
            implies x in E. Note that for the second direction, we always
            start with an input T(x) constructed by our algorithm, *not* an
            arbitrary input y for D (else we risk making the mistake
            described in the previous point).
```

```
2. (a) SubgraphIsomorphism in NP: Given G,H,c, where c maps vertices of G
       to vertices of H, check that H contains exactly the same edges as G
       on the mapped vertices.
       SubgraphIsomorphism is NP-hard: Clique -p> SubgraphIsomorphism --
       Given G,k, output G' = clique on k vertices, H = G.

   (c) MaxSat in NP: Given F,k,c, where c is an assignment of truth values,
       check that at least k clauses are satisfied.
       MaxSat is NP-hard: CNF-SAT -p> MaxSst -- Given F, output F,m where m
       = number of clauses of F.

   (e) SparseSubraph in NP: Given G,a,b,c, where c is a subset of vertices,
       check that c contains 'a' vertices that have no more than b edges
       between them.
       SparseSubgraph is NP-hard: IndependentSet -p> SparseSubgraph --
       Given G,k, output G,k,0.

   (f) SetCover in NP: Given S,S_1,...,S_m,k,c, where c (_ {1,...,m}, check
       that c contains k elements and that every element of S belongs to
       some subset selected by c.
       SetCover is NP-hard: VertexCover -p> SetCover -- Given G,k, output
       S=E, S_i = {all edges adjacent to v_i}, k.


3. The main ideas required are presented in the textbook, page 252, and in
   the lecture summary for last week. For reference, here is a detailed
   write-up of one of the reductions.

    *  VC -p> IS:

           On input (G,k) (for VC), construct (G',k') (for IS) as follows:
               Set G' = G and k' = n-k (where n = |V| in G).

       Clearly, (G',k') can be computed from (G,k) in polytime.
       Also, if G contains a vertex cover C of size k, then V - C is an
       independent set in G of size n-k: since every edge of G has at least
       one endpoint in C, no edge has both endpoints in V - C.
       Finally, if G contains an independent set I of size n-k, then V - I
       is a vertex cover of size k: since no edge of G has both endpoints
       in I, every edge of G has at least one endpoint in V - I.

       Common errors:

       "Given G,k, construct a new graph by taking the vertex cover in G
       and turning it into an independent set as follows: ..."
           The input for the reduction is *only* G and k: the reduction
           function does *not* have a vertex cover for G -- in fact, there
           may not even be one.
           And if you try to "fix" this by writing a transformation that
           searches for a vertex cover in G, then your algorithm will not
           run in polytime anymore (finding vertex covers is NP-hard).

       "Transform G,k into a new input G,n-k: if G contains a vertex cover
       of size k, then G contains an independent set of size n-k.  For the
       reverse, transform G,k into G,n-k again: if G contains an
       independent set of size k, then G contains a vertex cover of size
       n-k."
           This argument gives *two* transformations: one from VC to IS and
           a separate one from IS to VC (the fact that the same
```

transformation works in both directions is a side-effect of the relationship between the two problems here). And for each transformation, it only argues that the transformation preserves yes-instances.

"Transform G,k into G,n-k. If G contains a vertex cover of size k, then G contains an independent set of size n-k. If H is a graph that contains an independent set of size n-k, then we can construct the graph H,k that contains a vertex cover of size k."

The second half of the argument makes the same mistake as above, by describing a second construction. This happened because the argument was phrased in terms of a general input to the independent set problem, instead of considering only the specific input constructed by the reduction function.

Take the time to write up each of the other possibilities -- some of them will be nearly identical to each other because of the close relationship between the problems. Note that this is NOT true in general, for example, we don’t expect that a reduction from SubsetSum to 3SAT would look anything like the reduction from 3SAT to SubsetSum.