Template for proofs of NP-completeness: To show A is NPc, prove that

A in NP: Describe a polytime verifier for A.
"Given (x,c), describe the *type* of c and check that c has correct properties..."
Argue that verifier runs in polytime and that
x is a yes-instance iff verifier outputs "yes" for some c.

Note that all problems in NP we've seen so far have a similar structure to their definition: "the answer for object A is Yes iff there is some related object B such that some property holds about A and B" -- for example, for CLIQUE: "the answer for undirected graphs G and integers k is Yes iff there is a subset of vertices C that forms a k-clique in G". For all such problems, the verifier will also have a common structure: "on input (A,c), check that c encodes an object B and that A and B have the required property". Because of the way these decision problems are defined, this guarantees (A,c) is accepted for some c iff A is a yes-instance. All that remains is to ensure checking property of A,B can be done in polytime.

A is NP-hard: Show B -p> A for some NP-hard B.
"Given x, construct y_x as follows: ..."
Argue that construction can be carried out in polytime and that x yes-instance iff y_x yes-instance (often by showing x yes-instance => y_x yes-instance and y_x yes-instance => x yes-instance)
In more detail, this involves:
- starting with arbitrary input x for B (i.e., without making any assumption about whether x is a yes-instance or a no-instance),
- describing explicit construction of specific input y_x for A,
- arguing construction can be carried out in polytime,
- arguing if x is a yes-instance, then so is y_x,
- arguing if y_x is a yes-instance, then so was x (or equivalently, if x is a no-instance, then so is y_x).

Watch last step! Argument starts from y_x constructed earlier (not from arbitrary input y for A), and relates it to arbitrary x that y_x was constructed from.

Traps to watch out for:
- Direction of reduction: start from arbitrary input x for B (cannot place any restrictions on input; reduction must work with all possible inputs) and explicitly construct specific input y_x for A.
- "Reduction" that does something different for yes-instances vs. no-instances: this would involve telling the difference, which can't be done in polytime when B is NP-hard.
- "Reduction" that makes use of a certificate in the construction of y_x: when B is NP-complete, we know a certificate *exists* but that does not mean we can *find* it in polytime; also, reduction has to start from input x alone -- no certificate.

[NOTE: This example was NOT covered in class and is included here for your reference.]
One more example of reduction, to show interesting ideas: SAT -p> 3SAT.

- Input for reduction? Formula \phi.
  Output of reduction? Formula \phi' in 3-CNF.

Relationship? \phi satisfiable iff \phi' satisfiable.

- Trick: introduce new variable for each connective in \phi; write clauses that express "value of new variable = value of connective"; \phi' is conjunction of all these clauses together with clause z_1, where z_1 is new variable for main connective in \phi.

- Example: (x_1 /\ x_2) => ~x_3.
  Use z_1 for =>, z_2 for /\, z_3 for ~.
  \phi' = z_1 /\ (z_1 <=> (z_2 => z_3))
  /\ (z_2 <=> (x_1 /\ x_2))
  /\ (z_3 <=> ~x_3)

- Replace each "pseudoclause" (a <=> A) with up to eight clauses as follows: write down truth table for (a <=> A); for each line where (a <=> A) = False, write clause that is False for that line.
  For example: a <=> (b /\ c)

0	1	0	0	no clause (line is True)
0	1	0	1	no clause (line is True)
0	1	1	0	no clause (line is True)
0	0	1	1	clause: a \/ ~b \/ ~c
1	0	0	0	clause: ~a \/ b \/ c
1	0	0	1	clause: ~a \/ b \/ ~c
1	0	1	0	clause: ~a \/ ~b \/ c
1	1	1	1	no clause (line is True)

- Final \phi' = conjunction of each clause, repeating literals as needed in clauses with fewer than three literals (e.g., pseudoclause "z_1" becomes (z_1 \/ z_1 \/ z_1)).

- Runtime for reduction?
  O(m) -- scan all of \phi for each pseudoclause
  O(m) -- replace each pseudoclause by up to 8 clauses

- If \phi satisfiable, then assign values to new variables to match values of each connective in \phi -- this satisfies \phi'.
  If \phi' satisfiable, then by construction each new variable has value equal to some connective in \phi; because of clause z_1, this means \phi is satisfied.

Self-reducibility

NOTE: The summary in this section is more detailed than usual, because this material is not covered explicitly in the textbook (only showing up in exercises).

We've focused on decision problems, but many problems are more naturally "search problems": given input x, find solution y.

Examples:
- Given prop. formula F, find satisfying assignment, if one exists.
- Given graph G, integer k, find a clique of size k in G, if one exists.
- Given graph G, find a Ham. path in G, if one exists.
- Given set of numbers S, target t, find subset of S whose sum equals t, if one exists.
- etc.

Notation: We say A is "Turing-reducible" to B in polynomial time iff:

  . _assuming_ algorithm S_B solves B in constant time
    (even when we know there is no such algorithm),
  . we can write an algorithm S_A to solve A
    (by making appropriate calls to S_B), where
  . S_A runs in worst-case polynomial time.
This does _not_ require that B is actually solvable in polynomial-time. It is a conditional result.

Clearly, efficient solution to search problem would give efficient solution to corresponding decision problem. So proof that decision problem is NP-hard implies that search problem is "NP-hard" as well (in some generalized sense of NP-hard), and does not have efficient solution.

Example: Clique-Decision is Turing-reducible to Clique-Search:
  - Suppose CLS(G,k) returns a clique of size k in G (or NIL),
    in polytime.
  -     CLD(G,k):
            return CLS(G,k) != NIL
    solves Clique-Decision, also in polytime.

But exactly how much more difficult are search problems?

Perhaps surprisingly, many are only polynomially more difficult than corresponding decision problem, in the following sense: any efficient solution to the decision problem can be used to solve the search problem efficiently. In other words, the search problem is Turing-reducible to the decision problem! This is called "self-reducibility".

Example 1: CLIQUE-SEARCH
    Input: Undirected graph G, positive integer k.
    Output: A clique of size k in G, if one exists; special value NIL if
        there is no such clique in G.

  - Assumption: There is an algorithm CLD(G,k) that returns True iff G
    contains a clique of size k -- i.e., CLD solves the decision problem.

    WARNING! The argument for self-reducibility is NOT that "any algorithm that solves the decision problem must include a part that solves the search problem", as this is in fact not always true. For example, there is an algorithm that can determine whether or not an integer has any factors (i.e., whether or not it belongs to COMPOSITES) without actually finding any of the factors (through some fairly involved number theory about properties of prime numbers).

    In this case, for example, it would be a circular argument to assume that the algorithm CLD must find a clique of size k in order to return whether or not such a clique exists. Instead, what we must do is show how to write a different algorithm that searches for a k-clique in G, by making use of the information provided by calls to CLD, treating the CLD algorithm as a black-box -- all we know about it is that it produces the correct output for every instance of the decision problem, but we cannot make any assumption about how it does this.

  - Idea: For each vertex in turn, remove it iff resulting graph still
    contains a k-clique. Intuition: because output of CLIQUE-SEARCH is a subset of the input, this will remove everything we don't need.

  - Details: We construct an algorithm to solve CLIQUE-SEARCH as follows.

    CLS(G,k):

```
    if not CLD(G,k):  return NIL  # no k-clique in G
    for each vertex v in V:
        # remove v and its incident edges
        V' <- V - {v};  E' <- E - { (u,v) : u in V }
        # check if there is still a k-clique
        if CLD(G'=(V',E'),k):
            # v not required for k-clique, leave it out
            V <- V';  E <- E'
    return V
```

- Correctness: CLD(G=(V,E),k) remains true after every iteration of the main loop (it's a loop invariant) so at the end, V contains every vertex in a k-clique of G. At the same time, every other vertex will be taken out because it is not required, so V will contain no other vertex. Hence, the value returned is a k-clique of G.

- Runtime: Each vertex of G examined once, and one call to CLD for each one, plus linear amount of additional work (removing edges). Total is O((n+1)*t(n,m) + n*(n+m)) where t(n,m) is runtime of CLD on graphs with n vertices and m edges; this is polytime if t(n,m) is polytime.

- Exercise: What happens if G contains more than one k-clique?

General technique to prove self-reducibility:
- assume hypothetical algorithm to solve decision problem,
- write algorithm to solve search problem by making calls to decision problem algorithm (possibly many calls on many different inputs), treating decision algorithm as a black-box,
- argue search algorithm is correct, and
- make sure that search problem algorithm runs in polytime if decision problem algorithm does -- argue at most polynomially many calls to subroutine are made and at most polytime spent outside those calls.

Example 2: HAMPATH-SEARCH
Input: Graph G.
Output: A Ham. path in G.

- Idea 1: For each vertex in turn, remove it iff resulting graph still contains a Ham. path.

  Problem: Every vertex must be in the path anyway, and this does not say where to put each vertex (which edges to use to travel through this vertex).

- Idea 2: For each edge in turn, remove it iff resulting graph still contains a Ham. path -- same as for CLIQUE above, except considering edges one-by-one instead of vertices.

  Same algorithm, argument of correctness and runtime analysis as for CLIQUE, except looping over edges instead of vertices.

Example 3: VERTEX-COVER-SEARCH
Input: Graph G, integer k.
Output: A vertex cover of size k, if one exists (NIL otherwise).

- Idea 1: Remove vertices one-by-one as long as resulting graph still contains a vertex cover of size k.

- Problem: If G contains a VC of size k, then G-v (remove v and all incident edges) also contains a VC of size k, whether or not v is in the

cover (unless n < k, trivial to solve)!

- Idea 2: Check if G-v contains a VC of size (k-1).

- Algorithm:

```
VCS(G,k):
    if not VC(G,k):  return NIL
    C <- {}  # the vertices in a vertex cover of G
    for each vertex v in V, and while k > 0:
        if VC(G-v, k-1):
            C <- C U {v};  G <- G - v;  k <- k - 1
    return C
```

- Correctness: Loop invariant: G contains a VC D of size k and C u D is a vertex cover of size k_0 in G_0. At each iteration,
  . if G-v contains a VC of size (k-1), then G contains a VC of size k that includes v: say C' is VC of size (k-1) in G-v, then C' u {v} is a VC of size k in G;
  . if G contains a VC C of size k that includes v, then G-v contains a VC C - {v} of size (k-1); taking the contrapositive: if G-v does not contain a VC of size (k-1), then v does not belong to any VC of size k in G.

- Runtime: O((n+1)*t(n,m) + n*(n+m)) -- for each vertex v, we perform one call to VC in time t(n,m) and compute G-v in time O(n+m).

Optimization problems:

- Some search problems with one or more numerical parameters naturally occur in practice in the form of optimization problems, e.g.,

  . MAX-CLIQUE
  . MIN-VERTEX-COVER
  . etc.

  Optimization problems can also be polytime self-reducible by using decision problem algorithm to find optimal value of relevant parameter(s), then doing self-reduction to the search problem.

  Example: MAX-CLIQUE
  Idea: Given G, perform binary search in range [1,n] by making calls to CLD(G,k) for various values of k, in order to find value k such that G contains some k-clique but no (k+1)-clique. Then, use search algorithm to find k-clique. This makes O(log n) calls to CLD in addition to time to find the clique. (Note: Linear search for value of k would also be OK in this case because search is in the range [1..n] and input size = n. But in general, binary search is "safer", especially when working on a range of values related to numerical parameters.)

  Similar idea would work for MIN-VERTEX-COVER, etc.

For Next Week
* Readings: Sections 9.2 (intro), 9.2.1, 9.2.2
* Self-Test: Trace both approximation algorithms on a few inputs.