Decision problems, P, and NP:

* Decision Problem: problem D where solution is a single yes/no answer. e.g., "Given graph G, vertices s,t, is there a path from s to t?" "Given weighted graph G, bound b, is there a spanning tree of G with total weight <= b?"

* Why limit ourselves to decision problems?
  We want to prove negative results (that problems have no efficient solution). Intuitively, decision problems are "easier" than corresponding optimization/search problems, so if we can show that a decision problem is hard (has no efficient algorithm), this would imply that the more general problem is also hard. Also, turns out in most cases, search problem can be solved with the help of an algorithm for the decision problem. We'll make this precise later...

* Recall formal language = set of "strings" (finite sequences of characters over fixed "alphabet", often {0,1}), e.g.,
  { <G,s,t> | G is a graph that contains a path from s to t }
  where <G,s,t> represents any reasonable encoding of a graph and two vertices as a string (e.g., encoded using bits).

* Equivalence:
  For any decision problem D, define "language of D" as follows:
  L_D = { x | x encodes an input to D whose answer is "yes" }
  For any language L, define "acceptance problem for L" as follows:
  D_L = on input x, output "yes" if x in L, output "no" if x not in L

* From now on, slight abuse of notation: "x in D" means "x is a yes-instance of D" and "x not in D" means "x is a no-instance of D".

* The class P: All decision problems that have polytime algorithms.

* The class NP: All decision problems D that can be solved by a "generate-and-verify" algorithm with the following structure:

```
On input x:
    generate all "certificates" c, and for each one:
        # "verification" phase
        if verify(x,c):
            return True  # property holds for current certificate c
    return False  # property fails for every certificate c
```

  where the verification phase (the call verify(x,c)) runs in worst-case polynomial time, as a function of size(x). The time required to run the entire algorithm may be exponential but for NP, we care only about the time for the verify phase.

* Example: COMPOSITE (given positive integer x, does x have any factors?) belongs to NP because it is solved by generate-and-verify algorithm where generate phase loops over all integers c in {2,3,...,x-1} and verify(x,c):

```
return (c mod x == 0)
```

  Note: entire generate-and-verify algorithm equivalent to obvious algorithm to find a divisor:

```
for all integers c = 2,3,...,x-1:
    if c divides x:
```

```
            return True
        return False  # no divisor found
```

If x is composite, algorithm outputs True for some value of c. If x is not composite (i.e., x is prime), algorithm returns False. Moreover, verify(x,c) runs in polytime as a function of |x|: basic arithmetic operations are polytime.

Note: algorithm does *not* run in polynomial time overall! Runtime Theta(x) sounds good, except remember: size(x) = log_2 x. So as a function of n = size(x), runtime is Theta(2^n) -- exponential.

* Why the complicated definition? Doesn't correspond to any practical notion of computation. However, turns out to capture the computational complexity of a vast majority of "real-life" problems.

* Usually, NP defined in terms of "verifier": the verification phase in previous algorithm. In general, a verifier for decision problem D is an algorithm V that takes two inputs (x,c) such that:
  - for all yes-instances x (inputs to D for which the answer is yes), there is some string c such that V(x,c) outputs True in polytime (c is called a "certificate");
  - for all no-instances x, for all strings c, V(x,c) outputs False (no restriction on runtime).

  In other words, for all inputs x, the answer is yes iff there is a certificate c such that V(x,c) outputs True in polytime (measured as a function of n = |x| only, ignoring the size of c).
  Notice the asymmetry in the definition: it's important!

* Example: VERTEX-COVER
  Input: Undirected graph G, positive integer k
  Question: Does G contain a vertex cover of size k, i.e., a set C of k vertices such that each edge of G has at least one endpoint in C?

  Verifier for VERTEX-COVER:
  - On input <G,k,c>:
    verify that c is a subset of exactly k vertices of G
    check that c forms a vertex cover in G
    output True if both conditions hold; False otherwise
  - First step takes time O(kn) -- assuming we don't know anything about c and must check it is an actual subset of V; second step takes time O(mk); total is O(k(m+n)) = polynomial.
  - Verifier outputs True for some c iff G contains some VC of size k.

  For reference, here is the full "generate-and-verify" algorithm for VERTEX-COVER:

```
    On input <G,k>:
        for all subsets of vertices c:
            if c is a subset of exactly k vertices and
                    c forms a vertex cover:
                return True
        return False
```

  FROM NOW ON, OMIT "GENERATE-AND-VERIFY" AND PROVIDE ONLY VERIFIERS.

coNP:

* D in NP means there is a verifier V(x,c) running in polytime such that
  - V(x,c) = True for some c whenever x is a yes-instance,
  - V(x,c) = False for all c whenever x is a no-instance.

Notice the asymmetry: possible to verify yes-instances in polytime, but nothing known about no-instances.

* coNP = complements of problems in NP, i.e., problems whose NO-instances can be verified in polytime but for which we have no information about yes-instances.

* D in coNP iff there is a verifier V(x,c) running in polytime such that
  - V(x,c) = False for SOME c whenever x is a NO-instance,
  - V(x,c) = True for ALL c whenever x is a YES-instance.

* Example 1: Prime in coNP because Composite (= NonPrime) in NP.

* Example 2: DENSE problem:
  - Input: Undirected graph G = (V,E), positive integer k.
  - Output: Does _every_ subset of k vertices contain at least one edge between vertices in the subset?

  DENSE in coNP:

```
    On input G,k,c:
        if c is a subset of k vertices and
                G contains NO edge between any two vertices in c;
            return False
        else:
            return True
```

  Verifier runs in polytime and returns False for some c iff (G,k) not in DENSE.

  Note: DENSE is the "complement" of Independent-Set (IS):
  - In: Undirected graph G, positive integer k.
  - Out: Does G contain some independent set of size k (a subset of vertices with NO edge between any two vertices in the subset)?

Relationships:

* P subset of NP but NP != P has not been proven (yet)

* NP != coNP has not been proven (yet).

* P subset of coNP intersect NP, but unknown whether it is equal.

Polytime reductions/transformations:

* Formalize notion that one problem/language is "no harder" than another.

* Let I_1 = {all inputs to D_1}, I_2 = {all inputs for D_2}.
  D_1 -p> D_2 iff there is a function f : I_1 -> I_2 computable in polytime such that for all x in I_1, x in D_1 iff f(x) in D_2.

  In other words, inputs for D_1 can be transformed (in polytime) into inputs for D_2 such that the answers are the same for both inputs.

* We have seen many concrete examples of reductions, when working with network flows and linear programming. Now we apply reductions in a more abstract setting: not to actually solve problems but just to show that a certain relationship exists between them.

* Example: Independent-Set -p> Vertex-Cover.
  f(G,k) = (G,n-k) -- where n = $|V|$, as usual
  - Clearly computable in polytime.

- If G contains an ind. set S of size k, V-S is a vertex cover of size n-k, and the converse also holds.

* Theorem: If D_1 -p> D_2 and D_2 in P, then D_1 in P. (Same for NP).
  Proof: By definition, D_1 -p> D_2 means there is some polytime computable f : I_1 -> I_2 such that for all x in I_1, x in D_1 iff f(x) in D_2.
  D_2 in P means there is some algorithm A such that for all y in I_2, A(y) = True iff y in D_2.
  The following is a polytime algorithm for D_1:
      On input x, compute f(x) and return A(f(x)).
  Clearly, answer is correct: x in D_1 iff f(x) in D_2 iff A(f(x)) = True.
  Since f(x) computable in polytime, |f(x)| (the size of f(x)) is a polynomial in |x|, so runtime of A(f(x)) is a polynomial function of a polynomial in |x|, which is still polynomial.
  Same argument works with polytime verifier V(x,c) in place of algorithm A(x), to show the result for NP.

* Corollary: If D_1 -p> D_2 and D_1 not in P, then D_2 not in P.

NP-completeness:

* Use -p> to identify "hardest" problems in NP.

  Decision problem D is "NP-complete" iff:
   1. D in NP
   2. D is "NP-hard": for all D' in NP, D' -p> D.

* Theorem: If D is NP-complete, then D in P iff P = NP.
  Proof:
   <= If P = NP, then D NPc implies D in NP implies D in P.
   => If D in P, then D NPc implies D NP-hard implies for all D' in NP, D' -p> D so D' in P since D in P. Hence, NP subset of P so NP = P.

* Corollary: If P != NP and D is NP-complete, then D not in P.

For Next Week
* Readings: Sections 8.1, 8.3 (the first five reductions)
* Self-Test: Exercise 8.10(a),(c),(e)