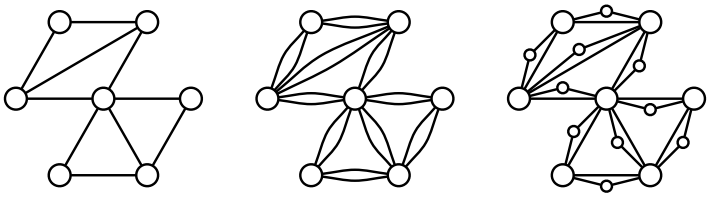
1. (a) Describe a polynomial-time reduction from EVENGRAPHISOMORPHISM to GRAPHISOMORPHISM.
   (b) Describe a polynomial-time reduction from GRAPHISOMORPHISM to EVENGRAPHISOMORPHISM.
   (c) Describe a polynomial-time reduction from GRAPHISOMORPHISM to SUBGRAPHISOMORPHISM.
   (d) Prove that SUBGRAPHISOMORPHISM is NP-complete.
   (e) What can you conclude about the NP-hardness of GRAPHISOMORPHISM? Justify your answer.

Solution:

(a) The reduction is trivial: Given two graphs G and H in which every vertex has even degree, we can determine whether they are isomorphic simply by calling GRAPHISOMORPHISM(G, H).

> **Rubric:** 1 point, all or nothing. No penalty for checking whether the graphs G and H are actually even, but it's not necessary.

(b) Suppose we are asked whether two graphs G and H are isomorphic. We can transform the input graphs into *even* graphs G' and H' in polynomial time by replacing each edge with a pair of parallel edges with the same endpoints. (Alternately, if parallel edges make you uncomfortable, we can replace each edge in the input graphs with a triangle, as shown below.) The new even graphs G' and H' are isomorphic if and only if the original graphs G and H are isomorphic. Thus, we can determine whether G and H are isomorphic by calling EVENGRAPHISOMORPHISM(G', H').



Transforming an arbitrary graph (left) into an even graph (middle or right).

> **Rubric:** 2 points max; no penalty for using multigraphs.

(c) Suppose we are asked whether two graphs G and H are isomorphic. We first check if the graphs have the same number of vertices and edges; if not, we immediately return FALSE. If the graphs do have the same size, then G is isomorphic to H if and only if G is isomorphic to a subgraph of H. Thus, we can determine whether G and H are isomorphic by calling SUBGRAPHISOMORPHISM(G, H). The reduction clearly takes polynomial time.

> **Rubric:** 2 points max

(d) We can easily verify in polynomial time that one graph is a subgraph of another if we are given the correspondence between vertices, so SUBGRAPHISOMORPHISM is in NP.

We prove the problem is NP-hard by a reduction from HAMILTONIANCYCLE. To determine whether a graph G contains a Hamiltonian cycle, we determine the number of vertices n, construct a cycle C_n of n vertices, and call SUBGRAPHISOMORPHISM(C_n, G). The reduction clearly takes polynomial time. Thus, SUBGRAPHISOMORPHISM is NP-hard.

We can also reduce from MAXCLIQUE. Let K_r denote the complete graph with r vertices.

```
MaxClique(G)
  for r ← 1 to ∞
      if SubgraphIsomorphism(K_r, G) = False
          return r − 1
```

The loop must stop before $r = n + 1$, so this reduction runs in polynomial time.

In fact, the reduction in the lecture notes implies that the following decision problem is NP-hard: Given a graph G with n vertices, does it contain a clique with $n/3$ vertices? We can solve this problem by calling SubgraphIsomorphism$(K_{n/3}, G)$.

Rubric: 3 points max = 1 for NP + 2 for NP-hard; these are not the only correct proofs.

(e) ***Nothing!*** The reduction is in the wrong direction!

Rubric: 1 point: all or nothing!

■

2. Suppose you are given a magic black box that can solve the 3Colorable problem ***in polynomial time***. Describe and analyze a ***polynomial-time*** algorithm that computes an actual proper 3-coloring of a given graph G, or correctly reports that no such coloring exists, using this magic black box as a subroutine.

Solution: The following algorithm runs in $O(n)$ time plus $O(n)$ calls to the magic black box.

```
MAGIC3COLOR(G):
  if MAGICBOX(G) = FALSE
      return NONE

  add new vertices r, g, b and edges rg, gb, br to G

  for every vertex v in G
      if MAGICBOX(G + vr + vg)
          add edges vr and vg to G
          color v blue
      else if MAGICBOX(G + vr + vb)
          add edges vr and vb to G
          color v green
      else ⟨⟨if MAGICBOX(G + vg + vb)⟩⟩
          add edges vg and vb to G
          color v red
```

The algorithm adds a triangle of new vertices r, g, b to the graph; these vertices represent the three colors red, green, and blue, respectively. The algorithm is clearly correct if G is *not* 3-colorable, so assume it is.

Consider an arbitrary vertex v. Any valid 3-coloring of G in which v is blue is also a valid 3-coloring of $G + vr + vg$; conversely, any valid coloring of $G + vr + vg$ is also a valid 3-coloring of G (in which v must be blue). Thus, the graph G is still 3-colorable after the first *if* block. Similar arguments imply that G is still 3-colorable after the other two *if* blocks. It follows by induction that the graph G is 3-colorable during the entire execution of the algorithm.

On the other hand, if G contains the edges vr and vg, vertex v *must* be blue, and similarly for the other two cases. Thus, the algorithm computes a valid 3-coloring of the final graph G (in fact, the *only* such 3-coloring), and this is also a valid 3-coloring of the original input graph. ■

Solution: The following algorithm runs in $O(n^2)$ time plus $O(n^2)$ calls to the magic black box.

```
MAGIC3COLOR(G):
  if MAGICBOX(G) = FALSE
      return NONE

  for every vertex u in G
      for every vertex v in G
          if uv is not an edge in G and MAGICBOX(G + uv)
              add edge uv to G

  for every vertex v in G
      if v has a red neighbor and a green neighbor in G
          color v blue
      else if v has a red neighbor in G
          color v green
      else
          color v red
```

The algorithm systematically tests each pair of vertices in G, and adds an edge between them if the resulting graph is still 3-colorable. After the first phase of the algorithm, G is still 3-colorable,

but adding any missing edge to G would make it no longer 3-colorable. In any 3-coloring of G, connected vertices must have different colors. On the other hand, if G had a 3-coloring in which two disconnected vertices have different colors, the edge between them should have been added by the algorithm. Thus, in any 3-coloring of the final graph G, two vertices have different colors *if and only if* they are connected. The second phase of the algorithm assigns colors greedily according to this rule. ■

Rubric: 10 points max = 6 for algorithm + 3 for proof of correctness + 1 for "polynomial time". These are not the only correct solutions. These proofs are longer than necessary for full credit.

3. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-complete.

Solution: We can easily verify in polynomial time whether a given permutation of the vertices is a heavy Hamiltonian cycle, so the problem is in NP.

We show that this problem is NP-hard by describing a reduction from the Hamiltonian ***path*** problem. The input to the Hamiltonian path problem is an undirected graph G (*without* weights on the edges). We modify G in polynomial time to obtain another edge-weighted graph H, as follows:

- Add two new vertices s and t.
- Add edges from s and t to all the other vertices (including each other).
- Assign weight 1 to the edge st and weight 0 to every other edge.

The total weight of all edges in H is 1. Thus, a Hamiltonian cycle in H is heavy if and only if it contains the edge st.

I claim that H contains a heavy Hamiltonian cycle if and only if G contains a Hamiltonian path. This claim can be proved as follows:

$\Longrightarrow$ First, suppose G has a Hamiltonian path from vertex u to vertex v. By adding the edges vs, st, and tu to this path, we obtain a Hamiltonian cycle in H. Moreover, this Hamiltonian cycle is heavy, because it contains the edge st.

$\Longleftarrow$ On the other hand, suppose H has a heavy Hamiltonian cycle. This cycle must contain the edge st, and therefore must visit all the other vertices in H contiguously. Thus, deleting vertices s and t and their incident edges from the cycle leaves a Hamiltonian path in G.

This reduction clearly requires only polynomial time. ■

Rubric: 10 points max = 1 for NP proof + 3 for correct reduction + 1 for "polynomial time" + 2½ for 'if' proof + 2½ for 'only if' proof. This is not the only correct reduction.

4. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.

   Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

Solution: We show that this puzzle is NP-hard by describing a reduction from 3SAT.

Let Φ be a 3CNF boolean formula with m variables and n clauses. We transform this board into a puzzle configuration in polynomial time as follows. The size of the board in $n \times m$. For all indices i and j, we place a stone at position (i, j) as follows:

- If the variable x_j appears in the ith clause of Φ, we place a blue stone at (i, j).
- If the negated variable $\overline{x_j}$ appears in the ith clause of Φ, we place a red stone at (i, j).
- Otherwise, we leave cell (i, j) blank.

We claim that this puzzle has a solution if and only if Φ is satisfiable. This claim immediately implies that solving the puzzle is NP-hard. We prove our claim as follows:

$\Longrightarrow$ First, suppose Φ is satisfiable, and consider an arbitrary satisfying assignment. For each index j, remove stones from column j according to the value assigned to x_j:

- If $x_j = \textsc{True}$, remove all red stones from column j.
- If $x_j = \textsc{False}$, remove all blue stones from column j.

In other words, remove precisely the stones that correspond to FALSE literals. Because every variable appears in at least one clause, each column now contains stones of only one color (if any). On the other hand, each clause of Φ must contain at least one TRUE literal, and thus each row still contains at least one stone. We conclude the that puzzle is satisfiable.

$\Longleftarrow$ On the other hand, suppose the puzzle is solvable; consider an arbitrary solution. For each index j, assign a value to x_j depending on the colors of stones left in column j:

- If column j contains blue stones, set $x_j = \textsc{True}$.
- If column j contains red stones, set $x_j = \textsc{False}$.
- If column j is empty, set x_j arbitrarily.

In other words, assign values to the variables so that the literals corresponding to the remaining stones are all TRUE. Each row still has at least one stone, so each clause of Φ contains at least one TRUE literal, so this assignment makes $\Phi = \textsc{True}$. We conclude that Φ is satisfiable.

This reduction clearly requires only polynomial time. ■

Rubric: 10 points max = 3 for correct reduction + 1 for "polynomial time" + 3 for 'if' proof + 3 for 'only if' proof. This is not the only correct reduction.

5. A boolean formula in ***exclusive-or conjunctive normal form*** (XCNF) is a conjunction (AND) of several *clauses*, each of which is the *exclusive*-or of one or more literals. The XCNF-SAT problem asks whether a given XCNF boolean formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-SAT or prove that it is NP-complete.

Solution: Any XCNF formula can be interpreted as a system of linear equations mod 2 as follows. Each variable is either 0 (false) or 1 (true), and the negation $\overline{x}$ of any variable x has value $1 - x = (1 + x) \bmod 2$. A clause is satisfied if and only if it contains an odd number of true literals, or equivalently, if the sum of the numerical values of the literals is equivalent to 1 (mod 2). The entire formula is satisfiable if and only if the resulting system of mod-2 linear equations has a solution.

We can solve the system of linear equations using Gaussian elimination, exactly as we would solve a system of linear equations over the reals, only using mod-2 arithmetic everywhere. In particular, if the system has no solution, Gaussian elimination will correctly report this fact. (Gaussian elimination works here precisely because the set $\{0, 1\}$ with mod-2 arithmetic is a field.) Given a system of m equations (clauses) in n unknowns (variables), Gaussian elimination requires $O(n^2 m)$ time. ■

Solution: Using the identity $(x \oplus \overline{y}) = \overline{(x \oplus y)}$, we can transform each clause into one of two standard forms: either the exclusive-or of one or more variables, or the negation of the exclusive-or of one or more variables:

$$(a \oplus b \oplus \cdots \oplus z) \quad \text{or} \quad \overline{(a \oplus b \oplus \cdots \oplus z)}$$

This transformation takes $O(nm)$ time, where n is the number of variables and m is the number of clauses.

Choose an arbitrary variable x. If x or $\overline{x}$ is a clause by itself, assign the only possible value to x, simplify the formula, and recurse. Otherwise, if x appears in more than one clause, we can reduce the number of clauses that contain x using one of the following identities (where y and z are subclauses):

$$\begin{aligned}
(x \oplus A) \wedge (x \oplus B) &= (x \oplus A) \wedge \overline{(A \oplus B)} \\
(x \oplus A) \wedge \overline{(x \oplus B)} &= (x \oplus A) \wedge (A \oplus B) \\
\overline{(x \oplus A)} \wedge \overline{(x \oplus B)} &= \overline{(x \oplus A)} \wedge \overline{(A \oplus B)}
\end{aligned}$$

(In these examples, A and B represent exclusive-ors of one or more variables.) By applying these rules repeatedly, we can ensure that some variable x appears in exactly one nontrivial clause. Recursively compute a satisfying assignment for all the other clauses. (If the other clauses cannot be satisfied, neither can the original formula.) The values of the other variables determine a unique value for x that satisfies its clause. The recursion bottoms out either when the formula is empty (and therefore vacuously satisfied) or contains both a variable x and its negation $\overline{x}$ as clauses (so the formula is not satisfiable).

Suppose the formula has m clauses and n variables. For each variable, we apply the simplification rules at most m times (once for each clause), and each application takes $O(n)$ time (constant time per variable). Once we have a satisfying assignment for the other variables, we can compute an appropriate value for x in $O(n)$ time. Thus, the total running time of the algorithm is $O(n^2 m)$.

This algorithm is more commonly known as ***Gaussian elimination*** (over the finite field $\mathbb{Z}_2$). ■

Rubric: 10 points max: 7 for correct algorithm + 3 for running time analysis. -1 for reporting a running time of $O(n^3)$. (This time bound is correct but weak, if n is the total length of the input formula.) It is not necessary to describe the Gaussian elimination algorithm, although the second solution does exactly that. No points for trying to prove NP-hardness, ***especially*** if you also try to give a polynomial-time algorithm.