Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

Reduction: Suppose we have an algorithm to solve problem A, how to use it to solve problem B?

This has been and will continue to be a recurring theme of the class. Examples so far include

- Use SCC to solve 2SAT.
- Use LP to solve max flow.
- Use max flow to solve mincut.
- Use max flow to solve maximum bipartite matching.

In each case, we would transform the instance of problem B we want to solve into an instance of problem A that we can solve. Importantly, the transformation is efficient, say, in polynomial time.

Conceptually, a efficient reduction means that problem B is no harder than A. On the other hand, if we somehow know that B cannot be solved efficiently, we cannot hope that A can be solved efficiently.

To show that the reduction works, you need to prove (1) if there is a solution for an instance of problem A, there must be a solution to the transformed instance of problem B and (2) if there is a solution to the transformed instance of B, there must be a solution in the corresponding instance of problem A.

If there exists a polynomial reduction from problem A to problem B, problem B is at least as hard as problem A. From this, we can define complexity class which sort of gauge 'hardness'.
Complexity Definitions

- NP: a decision problem in which a potential solution can be verified in polynomial time.
- P: a decision problem which can be solved in polynomial time.
- NP-Complete: a decision problem in NP which all problems in NP can reduce to.
- NP-Hard: any problem which is at least as hard as an NP-Complete problem.

Prove a problem is NP-Complete
To prove a problem is NP-Complete, you must prove the problem is in NP and it is in NP-Hard. To do this, you must show there exists a polynomial verifier, and reduce an NP-Complete problem to the problem.

1 NP Basics

Assume A reduces to B in polynomial time. In each part you will be given a fact about one of the problems. What information can you derive of the other problem given each fact?

1. A is in **P**.
2. B is in **P**.
3. A is **NP**-hard.

4. B is **NP**-hard.

2 SAT and Integer Programming

Consider the 3SAT problem, where the input is a set of clauses and each one is a OR of 3 literals. For example, $(x_1 \vee \overline{x}_4 \vee \overline{x_7})$ is a clause which evaluated to true iff one of the literals is true. We say that the input is satifiable if there is an assignment to the variables such that all clauses evaluate to true. We want to decide whether the input is satifiable.

On the other hand, consider the 0-1 linear programming problem. The setup is exactly the same as LP, except that the optimization problem is allowed to have 0-1 constraints such as $x_i \in \{0, 1\}$.

Show how to use 0-1 linear programming to solve 3SAT.

3 Decision vs. Search vs. Optimization

The following are three formulations of the VERTEX COVER problem:

- As a *decision problem*: Given a graph G, return TRUE if it has a vertex cover of size at most b, and FALSE otherwise.
- As a *search problem*: Given a graph G, find a vertex cover of size at most b (that is, return the actual vertices), or report that none exists.
- As an *optimization problem*: Given a graph G, find a minimum vertex cover.

At first glance, it may seem that search should be harder than decision, and that optimization should be even harder. We will show that if any one can be solved in polynomial time, so can the others.

(a) Suppose you are handed a black box that solves VERTEX COVER (DECISION) in polynomial time. Give an algorithm that solves VERTEX COVER (SEARCH) in polynomial time.

(b) Similarly, suppose we know how to solve VERTEX COVER (SEARCH) in polynomial time. Give an algorithm that solves VERTEX COVER (OPTIMIZATION) in polynomial time.

4 California Cycle

Prove that the following problem is NP-hard

Input: A directed graph $G = (V, E)$ with each vertex colored blue or gold, i.e., $V = V_{\text{blue}} \cup V_{\text{gold}}$

Goal: Find a *Californian cycle* which is a directed cycle through all vertices in G that alternates between blue and gold vertices (Hint : Directed Rudrata Cycle)