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.

Solution: If A reduces to B, we know B can be used to solve A, which means B is at least as hard as A. As a result, if B is in **P**, we can say that A is in **P**, and if A is **NP**-hard, we can say that B is **NP**-hard. If A is in **P**, or if B is **NP**-hard, we cannot say anything about the complexity of B or A respectively.

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.

Solution: We construct an explicit 0-1 linear program given a 3SAT formula, and show that it's feasible iff the SAT formula is satifiable. Let the variables be $x_1, \cdots, x_n$ and we put constraint $x_i \in \{0, 1\}$. Now we have to translate boolean clauses like $(x_1 \vee \overline{x}_4 \vee \overline{x_7})$ into linear constraints. We tranform them systematically like this:

$$x_1 \vee \overline{x_4} \vee \overline{x_7} \quad \iff \quad x_1 + (1 - x_4) + (1 - x_7) \geq 1$$

That is, we replace every negated variable $\overline{x}_i$ in the literal as $1 - x_i$ and replace the OR by $+$. The constraint ensures that at least one of the literals is true. We perform this tranform for each clause and get a bunch of linear constraints over binary variables. (We do not need an objective here; we can just write $\max 1$.)

The observation is that any satifiable assignment corresponds to a feasible solution to the integer program, and vice versa. Hence, even just solving the feasibility problem of 0-1 linear programming suffices 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.

Solution:

(a) If given a graph G and budget b, we first run the DECISION algorithm on instance (G, b). If it returns "FALSE", then report "no solution".

If it comes up "TRUE", then there is a solution and we find it as follows:

- Pick any node $v \in G$ and remove it, along with any incident edges.
- Run DECISION on the instance $(G \setminus \{v\}, b-1)$; if it says "TRUE", add v to the vertex cover. Otherwise, put v and its edges back into G.
- Repeat until G is empty.

Correctness: If there is no solution, obviously we report as such. If there is, then our algorithm tests individual nodes to see if they are in any vertex cover of size b (there may be multiple). If and only if it is, the subgraph $G \setminus \{v\}$ must have a vertex cover no larger than $b-1$. Apply this argument inductively.

Running time: We may test each vertex once before finding a v that is part of the b-vertex cover and recursing. Thus we call the DECISION procedure $O(n^2)$ times. This can be tightened to $O(n)$ by not considering any vertex twice. Since a call to DECISION costs polynomial time, we have polynomial complexity overall.

Note: this reduction can be thought of as a greedy algorithm, in which we discover (or eliminate) one vertex at a time.

(b) Binary search on the size, b, of the vertex cover.

Correctness: This algorithm is correct for the same reason as binary search.

Running time: The minimum vertex cover is certainly of size at least 1 (for a nonempty graph) and at most $|V|$, so the SEARCH black box will be called $O(\log |V|)$ times, giving polynomial complexity overall.

Finally, since solving the optimization problem allows us to answer the decision problem (think about why), we see that all three reduce to one another!

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)

Solution: We reduce Directed Rudrata Cycle to Californian Cycle, thus proving the NP-hardness of Californian Cycle. Given a directed graph $G = (V, E)$, we construct a new graph $G' = (V', E')$ as follows:

- For each $v \in V$, create a blue node v_b with an edge to a gold node v_g (in G').
- For each $(u, v) \in E$, add edge (u_g, v_b) to E'. Another way to view this is that for each node $v \in V$, we are redirecting all its incoming nodes to v_g, and all its outgoing nodes originate from v_b (in G').