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.

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 or not NP, that is the question

For the following questions, circle the (unique) condition that would make the statement true.

(a) If B is **NP**-complete, then for any problem $A \in$ **NP**, there exists a polynomial-time reduction from A to B.

Always True True iff $\mathbf{P} = \mathbf{NP}$ True iff $\mathbf{P} \neq \mathbf{NP}$ Always False

Solution: Always True: this is the definition of **NP**-hard, and all **NP**-complete problems are **NP**-hard

(b) If B is in **NP**, then for any problem $A \in$ **P**, there exists a polynomial-time reduction from A to B.

Always True True iff $\mathbf{P} = \mathbf{NP}$ True iff $\mathbf{P} \neq \mathbf{NP}$ Always False

Solution: Always true: since we have polynomial time for our reduction, we have enough time to simply solve any instance of A during the reduction.
Note that in this class, we ignore decision problems which always returns YES, and the decision problems which always returns NO.

(c) 2 SAT is **NP**-complete.

Always True True iff $\mathbf{P} = \mathbf{NP}$ True iff $\mathbf{P} \neq \mathbf{NP}$ Always False

Solution: True iff $\mathbf{P} = \mathbf{NP}$:
By definition, in order to be NP-Complete a problem must be in NP, and there must exist a polynomial reduction from every problem in NP.
If $\mathbf{P} \neq \mathbf{NP}$, then there does not exist a polynomial time reduction from NP-Complete problems like 3-SAT to 2-SAT.
If $\mathbf{P} = \mathbf{NP}$, then a polynomial reduction is as follows:
since $\mathbf{P} = \mathbf{NP}$ there must exist a polynomial times algorithm to solve $3 - SAT$. Thus, when we

are preprocessing 3-SAT we can solve for whether there exists a solution in the instance or not. If the instance has a solution, then we will map it to an instance of $2-SAT$ that has a solution, and if it doesn't have a solution, we will map it to an instance that doesn't have a solution. Thus all problems in NP will have a polynomial time reduction to 2-SAT as all problems in NP are reducible to 3-SAT.

(d) Minimum Spanning Tree is in **NP**.

Always True   True iff $\mathbf{P} = \mathbf{NP}$ True iff $\mathbf{P} \neq \mathbf{NP}$ Always False

Solution: Always True. MST is solvable in polynomial time, which means it is verifiable in polynomial time.
Note that explicitly, the decision problem would be "does there exist a spanning tree whose cost is less than a budget b?".

2 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').

3 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? Each part should be considered independent; i.e., you should not use the fact given in part (a) as part of your analysis of part (b).

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.

4 Local Search for Max Cut

Sometimes, local search algorithms can give good approximations to NP-hard problems. In the Max-Cut problem, we have an unweighted graph $G(V, E)$ and we want to find a cut (S, T) with as many edges "crossing" the cut (i.e. with one endpoint in each of S, T) as possible. One local search algorithm is as follows: Start with any cut, and while there is some vertex $v \in S$ such that more edges cross $(S - v, T + v)$ than (S, T) (or some $v \in T$ such that more edges cross $(S + v, T - v)$ than (S, T)), move v to the other side of the cut. Note that when we move v from S to T, v must have more neighbors in S than T.

(a) Give an upper bound on the number of iterations this algorithm can run for (i.e. the total number of times we move a vertex).

(b) Show that when this algorithm terminates, it finds a cut where at least half the edges in the graph cross the cut.

Solution:

(a) $|E|$ iterations. Each iteration increases the number of edges crossing the cut by at least 1. The number of edges crossing the cut is between 0 and $|E|$, so there must be at most $|E|$ iterations.

(b) $\delta_{in}(v)$ be the number of edges from v to other vertices on the same side of the cut, and $\delta_{out}(v)$ be the number of edges from v to vertices on the opposite side of the cut. The total number of edges crossing the cut the algorithm finds is $\frac{1}{2}\sum_{v \in V} \delta_{out}(v)$, and the total number of edges in the graph is $\frac{1}{2}\sum_{v \in V}(\delta_{in}(v) + \delta_{out}(v))$. We know that $\delta_{out}(v) \geq \delta_{in}(v)$ for all vertices when the algorithm terminates (otherwise, the algorithm would move v across the cut), so the former is at least half as large as the latter.

5 Cycle Cover

In the cycle cover problem, we have a directed graph G, and our goal is to find a set of directed cycles $C_1, C_2, \ldots C_k$ in G such that every vertex appears in exactly one cycle (a cycle cannot revisit vertices, e.g. $a \to b \to a \to c \to a$ is not a valid cycle, but $a \to b \to c \to a$ is), or declare none exists.

In the bipartite perfect matching problem, we have a undirected bipartite graph (a graph where the vertices can be split into L, R, and there are no edges between two vertices in L or two vertices in R), and our goal is to find a set of edges in this graph such that every vertex is adjacent to exactly one edge in the set, or declare none exists.

Give a reduction from cycle cover to bipartite perfect matching. (Hint: In a cycle cover, every vertex has one incoming and one outgoing edge.)

Solution: Given the cycle cover instance G, we create a bipartite graph G' where L has one vertex v_L for every vertex in G, and R has one vertex v_R for every vertex in G. For an edge (u, v) in G, we add an edge (u_L, v_R) in the bipartite graph. We claim that G has a cycle cover if and only if G' has a perfect matching.

If G has a cycle cover, then the corresponding edges in the bipartite graph are a bipartite perfect matching: The cycle cover has exactly one edge entering each vertex so each v_R has exactly one edge adjacent to it, and the cycle cover has exactly one edge leaving each vertex, so each v_L has exactly one edge adjacent to it.

If G' has a perfect matching, then G has a cycle cover, which is formed by taking the edges in G corresponding to edges in G': If we have e.g. the edges $(a_L, b_R), (b_L, c_R), \ldots, (z_L, a_R)$ in the perfect matching, we include the cycle $a \to b \to c \to \ldots, z \to a$ in G in the cycle cover. Since v_L and v_R are both adjacent to some edge, every vertex will be included in the corresponding cycle cover.