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.

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? 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).

(a) A is in **P**.

(b) B is in **P**.

(c) A is **NP**-hard.

(d) B is **NP**-hard.

Solution: If A reduces to B, we know that an algorithm for B can be used to solve A; this implies that solving B is at least as hard as solving A. By this, we mean that the time it takes to solve B must be at least the amount of time it takes to solve A.

(a) If A is in **P**, then we know that B must be a problem that is at least as hard as any problem in **P**. Thus, B can either be in **P**, **NP**, or **NP**-hard. Thus, we cannot derive any information about B here.

(b) If B is in **P**, we know that the difficulty of solving A is at most that of B. Thus, A is also in **P**.

(c) If A is **NP**-hard, then B must also be in **NP**-hard because it is at least as hard as A.

(d) If B is **NP**-hard, then A can be in **P**, **NP**, or **NP**-hard. Thus, we cannot derive any information about A here.

2 Exact 4-SAT

The Exact 4-SAT problem is defined as follows.

Input: n boolean variables $\{x_1, \dots, x_n\}$ and clauses $\{C_1, \dots, C_m\}$ with each containing exactly {four distinct} literals. For example, the following is an instance of Exact 4-SAT,

$$(x_1 \vee x_2 \vee \overline{x_4} \vee \overline{x_5}) \wedge (x_3 \vee \overline{x_4} \vee \overline{x_1} \vee x_2) \wedge (x_1 \vee x_3 \vee x_4 \vee x_5)$$

Note that all the 4 literals within a clause have to be distinct.

Goal: Find an assignment to the variables $x_1, \dots, x_n$ that satisfies all the clauses.

(a) Give a polynomial time reduction from 3-SAT to Exact 4-SAT.

(b) Give a polynomial time reduction from Exact 4-SAT to 3-SAT.

Solution: When approaching reductions (from A to B), the two main steps are:

1. Find the easiest way to convert an instance of A into an instance of B.
2. Figure out how to modify what you did in step 1 to ensure that the solution to the B instance can be properly used for the A instance.

Now, let's walk through the reductions!

(a) Given a 3-SAT instance, we want to find the easiest way to convert it into a 4-SAT instance. We can do this by creating a single additional variable y and add it to every clause. For example, for the 3-SAT clause, $(x_1 \vee x_2 \vee \overline{x_4})$, we create the 4-SAT clause $(x_1 \vee x_2 \vee \overline{x_4} \vee y)$. Does this reduction work?

No! And this is because calling the 4-SAT solver on this instance we've constructed will just set y to be true, and all clauses in the 4-SAT instance will be satisfied, even if the original 3-SAT instance wasn't satisfiable. So, how can we fix this?

The idea is that we want the original 3-SAT clauses to still "hold weight" in the transformed 4-SAT instance, and one way we can do this is to create two 4-SAT clauses for each 3-SAT clause, one with y added and the other with $\overline{y}$ added. For example, for the 3-SAT clause, $(x_1 \vee x_2 \vee \overline{x_4})$,we create the two 4-SAT clauses $(x_1 \vee x_2 \vee \overline{x_4} \vee y) \wedge (x_1 \vee x_2 \vee \overline{x_4} \vee \overline{y})$. We pass these clauses to the Exact 4-SAT solver.

Now, if the solver set y to be true, then half of the clauses containing y are immediately satisfied. Then, the other half containing $\overline{y} = \mathsf{FALSE}$ becomes equivalent to the input to the 3-SAT instance, so the assignment of x_i values must satisfy that input. This also happens if the solver set y to be false. Thus, in either case, the assignment of the x_i values must be a valid solution to the 3-SAT instance.

(b) To go from Exact 4-SAT to 3-SAT, we want convert each clause with 4 literals to clauses with at most 3 literals. To do this, for each 4-SAT clause, for example $(x_1 \vee x_2 \vee x_3 \vee x_4)$, create a new variable y. Then, add the corresponding clauses $(x_1 \vee x_2 \vee y) \wedge (x_3 \vee x_4 \vee \overline{y})$ to our 3-SAT instance.

Now, if the 4-SAT instance is satisfiable, then our $3-SAT$ instance will be satisfiable by setting y to true if the first two literals in that clause are false, and setting y to false otherwise.

Similarly, if such a 3-SAT instance is satisfiable, then since either y or $\overline{y}$ will be false, one of the literals in the original 4-SAT clause have to be true. Hence, the 4-SAT instance is also satisfiable.

3 Cycle Cover

In the cycle cover problem, we have a directed graph G, and our goal is to find a set of directed cycles $C_1, C_2, \dots C_k$ 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 $(a_L, b_R), (b_L, c_R), \dots, (z_L, a_R)$ in the perfect matching, we include the cycle $a \to b \to c \to \dots, z \to a$ 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.

4 Public Funds

You are looking to build a new fence for your mansion, to keep out pesky people protesting profligate purchases. You have m bank accounts at your disposal to use to pay for your fence; each account i has a balance of b_i. You must choose one of n options for your fence; each fence j costs c_j dollars. You would like to withdraw from at most k of the bank accounts to build the fence, and due to peculiar UC accounting rules, if you use a particular bank account, you must use the whole balance (all b_m dollars.)

Determine whether it is possible to exactly pay for some fence j; that is, whether there is a j between 1 and n such that you can withdraw exactly c_j dollars given the bank account balances $b_1, \ldots, b_m$, the fence costs $c_1, \ldots, c_n$, and k.

Your task is to prove that Public Funds is **NP**-complete.

(a) Prove that Public Funds is in **NP**.

(b) Prove that Public Funds is **NP**-hard by providing a reduction from Subset Sum.

Note: to rigorously prove the correctness of a reduction from A to B, you must show two things:

1. *If an instance of A has a solution, then the transformed instance of B has a solution.*
2. *If an instance of B in the format of the transformation has a solution, then the corresponding instance of A has a solution.*

Solution:

Proof that Public Funds is in NP.
We can verify a solution for Public Funds in polynomial time by verifying that the sum of the balances of the chosen bank accounts adds up exactly to the price of the fence.

Proof that Public Funds is in NP-Hard.
To show that it is in NP-Hard, we reduce from SUBSET SUM.

Let each of the m elements in the set of numbers correspond to a bank account's balance b_m. Let the desired sum s correspond to the price of the only fence c_1. Also, set the maximum number of bank accounts used, k, to the number of elements in the set of numbers m.

1. **If an instance of subset sum has a solution, then the transformed instance of Public Funds has a solution.**
If subset sum has a solution, then we can take the banks that correspond to the elements of the solution, and those banks should sum up to the only fence, thus Public Funds should have a solution.

2. **If an instance of Public Funds in the format of the transformation has a solution, then the corresponding instance of subset sum has a solution.**
If an algorithm for Public Funds finds a solution, we can interpret the bank accounts chosen as a solution for SUBSET SUM.

5 SAT and Integer Programming

Consider the 3-SAT 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 integer linear programming feasibility problem: We are given a set of variables and constraints in terms of these variables (we are not given an objective). The constraints are either linear inequalities, or the 0-1 constraints $x_i \in \{0, 1\}$. We want to decide if it possible to assign the variables values that satisfy all the constraints.

Give a reduction from 3-SAT to integer linear programming feasibility, and briefly justify its correctness. No runtime analysis needed.

Solution: We construct an explicit integer linear program given a 3-SAT formula, and show that it's feasible iff the SAT formula is satifiable.

Let the variables be $x_1, \ldots, x_n$ and we add the constraint $x_i \in \{0, 1\}$; here, setting $x_i = 1$ corresponds to setting x_i to true, with similar logic for $x_i = 0$.

Now we have to translate the boolean clauses like $(x_1 \vee \overline{x}_4 \vee \overline{x_7})$ into linear constraints. In order to do this, we take note that we want at least one of x_1, $\overline{x_4}$, or $\overline{x_7}$ to be true. In other words, we want $x_1 = 1$, $x_4 = 0$, or $x_7 = 0$ in the ILP instance. Thus, we transform the constraint systematically as follows:

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

That is, we replace every negated variable $\overline{x}_i$ in the literal as $1 - x_i$ and replace each OR operator with $+$. This constraint ensures that at least one of the literals is true, as at least one of x_1, $(1 - x_4)$, or $(1 - x_7)$ must be set to 1. We perform this tranform for each clause and get a bunch of linear constraints over binary variables.

As for the correctness, the main observation is that any satifiable assignment corresponds to a feasible solution to the integer program, and vice versa. Hence, solving the feasibility problem of integer linear programming suffices to solve 3SAT.

6 (Optional) Upper Bounds on Algorithms for NP Problems

(a) Recall the 3-SAT problem: we have n variables x_i and m clauses, where each clause is the OR of at most three literals (a literal is a variable or its negation). Our goal is to find an assignment of variables that satisfies all the clauses, or report that none exists.

Give a $O(2^n m)$-time algorithm for 3-SAT. Just the algorithm description is needed.

(b) Using part (a) and the fact that 3-SAT is **NP**-hard, give a $O(2^{n^c})$-time algorithm for every problem in **NP**, where c is a constant (that can depend on the problem). Just the algorithm description and runtime analysis is needed.

*Hint: Since 3-SAT is **NP**-hard, you can use it to solve any problem in **NP**!*

Note: This result is known as $NP \subseteq EXP$.

(c) Recall the halting problem from CS70: Given a program (as e.g. a .py file), determine if the program runs forever or eventually halts. Also recall that there is no finite-time algorithm for the halting problem. Let us define the input size for the halting problem to be the number of characters used to write the program.

Given an instance of 3-SAT with n variables and m clauses, we can write a size $O(n+m)$ program that halts if the instance is satisfiable and runs forever otherwise. So there is a polynomial-time reduction from 3-SAT to the halting problem.

Based on this reduction and part (b): Is the halting problem **NP**-hard? Is it **NP**-complete? Justify your answer.

Solution:

(a) Enumerate all 2^n assignments. We can check if each assignment is satisfying in $O(m)$ time. So we can find a satisfying assignment if one exists $O(2^n m)$-time.

(b) Any problem in NP can be reduced to an instance of 3-SAT in polynomial time. Since the reduction takes polynomial time, and it takes at least $O(1)$ time to write down a variable/clause, the 3-SAT instance can't have more than n^b variables and n^b clauses for some constant b (for all sufficiently large n). So our algorithm is to reduce to 3-SAT and then solve this 3-SAT instance in $O(2^{n^b} n^b) = O(2^{n^{b+1}})$ time using part a. This is $O(2^{n^c})$ for $c = b + 1$.

Alternate approach: Since the problem is in NP, if the answer to the problem is "yes", there is some polynomial-size "witness" to the problem that we can feed to a verification algorithm and have it output "yes" in polynomial time. If the answer is no, none of these witness cause it to output "yes".

So, we can enumerate over all witnesses, and feed them to this verification algorithm, and output yes if any run of the verification algorithm outputs yes. Since there are polynomially many witnesses, and the verification algorithm runs in polynomial time, this takes $O(2^{n^c})$ time for some constant c.

(c) The reduction implies the halting problem is NP-hard. We can show the halting problem isn't in NP, and thus it isn't NP-complete: By part (a), any problem in NP has a finite-time algorithm, but the halting problem doesn't have one. So the halting problem isn't NP-complete.