This homework is due by the start of class at 1PM on Thursday, Apr 15th. You must submit the homework via the course page on T-Square.

Homework Policies:

- Your work will be graded on correctness and clarity. Write complete and precise answers.
- You may collaborate with up to *three* other classmates on this problem set. However, you must *write your own solutions* in your own words and *list your collaborators.*
- You may portions of our two textbooks that we have covered. The purpose of these homeworks is NOT only to teach you the algorithms that we cover, but also to teach you how to problem-solve. Therefore using *ANY* outside resources like the internet, previous students etc. is *not* allowed!
- **Optional** questions are for you to practice and learn the basics before you advance. They may ask you to follow the steps of an algorithm covered in class, or review material from an earlier class. Solutions to these problems will not be graded, but you must be are able to do them, and you are responsible for material that is addressed by these optional questions.
- **Basic** questions *must be solved and written up alone*; you may not collaborate with others. These will help you build the framework needed to solve the harder questions. Please do these first, before meeting in groups, for both you and your group's benefit.
- **Group** questions may be solved in collaboration with up to ***three*** other classmates. However, you must still *write up your own solutions* alone and in your own words. These will form the bulk of your homework questions.
- **Bonus** questions are like group questions, but they will be more challenging and will be graded more rigorously. Bonus points are tallied separately from your normal points, and will be applied to your grade *after* the curve. Bonus points are worth more than normal points, and can have a significant positive affect on your grade. Most importantly, bonus problems are fun!

Directions for Homework 7: You may assume that the following problems (both the search and decision versions) are NP-complete when doing your reductions.

- SAT : Given a boolean formula F in CNF form (ANDs of ORs) with n variables and m clauses, determine if there is assignment of the n variables that causes F to evaluate to $True$.
- $3-$SAT : A special case of SAT where every clause has at most 3 terms.
- Independent Set : Given graph G and number k, determine if G has an independent set of size $\geq k$. An independent set is a set of vertices such that no two share an edge.
- IntegerLinearProgramming : Given a set of n variables and m linear constraints, and a linear objective function, find an *integer* solution to the constraints that maximizes (or minimizes) the objective.

Remember that to show a problem A is NP-Complete, you need to do three things.

- Show that problem A is in NP. (Describe a verification algorithm for A).
- Reduce A from a previously established NP-hard problem B.

- First, describe an algorithm to solve the NP-hard problem B that uses a solution for A as a subroutine, and does polynomial amounts of work, and calls A polynomially many times on polynomial-sized inputs.
- Prove that your reduction is correct. You need to prove that your proposed algorithm for B will "accept" an input IF AND ONLY IF the input was a valid "yes" instance. This usually involves some analysis on how you transformed your input to problem B into the input to problem A in your reduction.

If problem A is a generalization of a known NP-complete problem, then the reduction step is much easier - just explain how to set the parameters of the more general problem to deal with the special case.

You do not have to write full pseudocode when describing your reductions, but you may if you feel that it will help you communicate your ideas. We expect your writing to be as clear as if you were writing pseudocode. A programmer should be able to read your description and implement your algorithm unambiguously.

1. (20 points) (basic) Show that the following problems, defined below, are in NP. (Do not show at this time that they are NP-complete).

   (a) Vertex Cover : Given graph G and number k, determine if G has a vertex cover of size $\leq k$. A vertex cover is a set of vertices such that every edge in G is incident to one of the chosen vertices.

   Solution: On a "yes" instance (G, k), let the certificate be the set of vertices in a vertex cover of size $\leq k$. Our verification algorithm will check that the certificate

   - Has fewer than k vertices in $O(k) = O(n)$ time.
   - Is indeed a vertex cover by iterating through every edge e in G, and checking that e has some endpoint in the vertex cover by iterating though the list of vertices in the certificate in $O(k|E|)$ time. (This can be done faster, but w/e - any polynomial will do).
   - Accept if both of the above are true.

   Our verification algorithm will return true iff the certificate corresponds to a vertex cover of size $\leq k$ in G. Such a certificate exists iff G has a vertex cover of size $\leq k$.

   (b) Clique : Given graph G and number k, determine if G has a clique of size $\geq k$. A clique is a set of vertices such that every pair of vertices share an edge.

   Solution: On a "yes" instance (G, k), let the certificate be the set of vertices in a clique of size $\geq k$. Our verification algorithm will check that the certificate

   - Has at least k vertices in $O(k) = O(n)$ time.
   - Is indeed a clique by iterating through every pair of vertices (v_1, v_2) in the certificate, and checking that that edge is present in the graph. Depending on your graph representation, this could take $O(1)$ time per pair for $O(k^2)$ total runtime, or even $O(|E|)$ time per pair for $O(k^2|E|)$- it doesn't matter, this is all still polynomial.

- Accept if both of the above are true.

Our verification algorithm will return true iff the certificate corresponds to a clique of size $\geq k$ in G. Such a certificate exists iff G has a clique of size $\geq k$.

(c) Hamiltonian Cycle (Rudrata) : Given graph G with n vertices, determine if G has a cycle of length n that visits every vertex exactly once.

Solution: On a "yes" instance (G, k), let the certificate be the set of vertices in a Hamiltonian Cycle presented in order of the cycle. Our verification algorithm will check that the certificate

- Has exactly $|V|$ vertices and visits every vertex exactly once in $O(|V|)$ time.
- Is indeed a cycle by checking that every pair (v_i, v_{i+1}) is an edge in the graph. Depending on the representation, this could take $O(V)$ or $O(V^2)$ time - it doesn't matter, this is all still polynomial.
- Accept if both of the above are true.

Our verification algorithm will return true iff the certificate corresponds to a cycle of size $|V| = n$ in G. Such a certificate exists iff G has a cycle of size $|V|$.

(d) Subset Sum : Given an array of integers A and a target T, determine if some subset of A adds up to T.

Solution: On a "yes" instance (A, T), let the certificate be the set of indices I of A that correspond to numbers that sum to T. Our verification algorithm will check:

- That $\sum_{i \in I} A[i] = T$.
- Accept if the above is true.

Our verification algorithm will return true iff the certificate corresponds to a subset of A that sums to T. Such a certificate exists iff A has a subset that sums to T.

2. (15 points) (group) Consider the following problem.
$\mathsf{SAT}_{\leq k}$: Given a SAT instance with n variables and m clauses, and an integer $k \leq n$, decide if there is a satisfying assignment in which *at most* k of the n variables are assigned TRUE. Prove that this problem is NP-complete.

Solution:

- First we show this is in NP. Let the certificate be the satisfying assignment that sets at most k variables ot true. Our verifier will check that
  - At most k of the variables are set to true. We can check this in $O(n)$ time.

- That this is a satisfying assignment by evaluating the formula in $O(mn)$ time.
- Accept if both of the above are True

This will accept iff there is a valid assignment

- We show that this problem is a generalization of SAT .

  Our reduction: On an input F to SAT , we return $\mathsf{SAT}_{\leq \mathsf{n}}(F)$.

  PROOF: Since the restriction "at most n of the variables are set to True" applies to any assignment, the problem $\mathsf{SAT}_{\leq n}$ is exactly the problem of $\mathsf{SAT}(F)$. ■

3. (15 points) (group) Consider the following problem.
Weighted Independent Set: Given a graph $G = (V, E)$ with integer weights on the vertices V, and number k, determine if G has an independent set of total weight $\geq k$.
Prove that this problem is NP-complete.

Solution: This was done in class, so you guys better get this one right!

- First we show this is in NP. Let the certificate be the independent set with total weight $\geq k$. Our verifier will check that
  - The given vertices have total weight $\geq k$ in at most $O(n)$ time.
  - That this is a valid independent set, by checking each edge and making sure that both endpoints are never in the set. This will take $O(E)$ time.
  - Accept if both of the above are True

  This will accept iff the certificate is a valid independent set of total weight $\geq k$, which exists iff there is an independent set of total weight $\geq k$.

- We show that this problem is a generalization of Independent Set .

  Our reduction: Given an instance (G, k) of Independent Set, create a weighted version of G where every vertex has weight 1. Call this graph G'. Now, return Weighted Independent Set(G', k).

  PROOF: By construction, any set of k vertices in G has total weight k in G' and vice versa. Thus an independent set of k vertices exists in G iff that independent set has total weight k in G'. ■

4. (30 points) (group) Self-Reducibility of SAT. Describe an algorithm that **finds** a satisfying assignment for a given SAT instance (search), if one exists, using an algorithm that can only determine **whether or not** a SAT instance has a satisfying assignment (decision).

Solution: Our algorithm will do the following. We'll call SAT_D the decision problem that will return True if the input assignment is satisfiable, and False otherwise.

We'll define a partial assignment of variables to a formula to be the formula that you would get if you plugged the values of the variables in your assignment into your formula. For example, if we did the partial assignment $(x_1 = True)$ to the formula $F = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$, we would get simply $F(x_1 = True) = (x_3 \vee x_4)$. For a few more examples, $F(x_1 = False) = (x_2) \wedge (x_3 \vee x_4)$, and $F(x_1 = False, x_2 = False) = False$. It's easy to see that we can construct a partial assignment of a formula in polynomial time by iterating over all the terms in the formula.

On input formula F, we start by running $\mathsf{SAT}_D(F)$. If this is False, we can say that a satsifying assignment doesn't exist, and return False or Null right away. Otherwise, there is a satisfying assignment to find. We'll use the following recursive strategy.

We start by running $\mathsf{SAT}_D(F(x_1 = True))$.

- If this is True, then there is some satisfying assignment of F where x_1 is True. So we will assign x_1 to True, and recurse on the simpler formula $F(x_1 = True)$ to find a satisfying assignment of the other variables.
- If it is False, then since there was a satisfying assignment somehow, there must be a satisfying assignment where x_1 is False. Thus we will set x_1 to False, and recurse on $F(x_1 = False)$ to find a satisfying assignment of the other variables.

This will require $O(n)$ constructions of these partial formulas $F(\ldots)$, each of which will take at most $O(mn)$ time. We also make $O(n)$ calls to SAT_D on inputs at most as large as the original imput to the problem. Thus this will take polynomial time!

5. (20 points) (group) The $\mathsf{Exact} - 3 - \mathsf{SAT}$ problem is a special case of $3 - \mathsf{SAT}$ where each clause has **exactly 3 terms**. Prove that $\mathsf{Exact} - 3 - \mathsf{SAT}$ is NP-complete.

Solution: This problem is in NP as this problem is a special case of $3 - \mathsf{SAT}$, which is in NP (A polytime verification algorithm for $3 - \mathsf{SAT}$ would also work as verification here.)

We reduce from $3 - \mathsf{SAT}$. Given an instance of $3 - \mathsf{SAT}$ with n variables and m clauses, we create an instance of Exact $3 - \mathsf{SAT}$ as follows:

First create two dummy variable x and y.

For each clause with 3 terms in the $3 - \mathsf{SAT}$ instance, we create the exact same clause in the Exact $3 - \mathsf{SAT}$ instance.

For each clause with 2 terms $(a \vee b)$ in the $3 - \mathsf{SAT}$ instance, we create two new clauses $(a \vee b \vee x) \wedge (a \vee b \vee \bar{x})$. It's not hard to see that this is logically equivalent to the original $(a \vee b)$ - no matter what x is.

Similarly for a clause with 1 term (a), we need to create the 4 clauses $(a \vee x \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee \bar{x} \vee \bar{y})$, which is also logically equivalent to a no matter what x and y are.

As explained above, our constructed Exact-3-SAT forumala is logically equivalent to our input $3 - \mathsf{SAT}$ formula, no matter what the assignment of the dummy variables. Thus an

assignment of the constructed formula is satisfiable iff the original $3-\mathsf{SAT}$ formula was satisfiable.

Secondly, this construction can be done in polynomial time, because we do at most $O(m)$ work in creating the clauses, and end with a formula with at most $4m$ clauses, and $n+2$ variables. Both of these are polynomial, thus the input formula to $\mathsf{Exact}-3-\mathsf{SAT}$ will be polynomial in the original input. Because all steps in the reduction take polynomial time, this implies that a polytime algorithm for Exact 3SAT would result in a polytime algorithm for $3-\mathsf{SAT}$. Thus Exact 3SAT is NP-complete.

Now I didn't explicity state that you couldn't have duplicate terms in a formula, so I will begrudgingly give credit to answers that just "padded" their clauses with repeated terms.

6. (15 points) (bonus)

We can make variants of the $3-\mathsf{SAT}$ problem by retaining the same general setup, (n variables, m clauses, ≤ 3 terms per clause), but then changing the requirement needed to satisfy a clause. In $3-\mathsf{SAT}$, that requirement is that you need at least one term in each clause to be set to True. The $3-\mathsf{SAT}$ problem asks to find an assignment that can satisfy all clauses simultaneously.

Either show that these variants of $3-\mathsf{SAT}$ are NP-Complete, or give a polynomial time algorithm that solves them.

(a) All Or None $3-\mathsf{SAT}$ is a variant where we are determining if there is an assignment where the terms in each clause are equal to each other (the terms in each clause are either ALL True or ALL False).

Solution: You can solve this one in polynomial time. First create a graph on $2n$ nodes (one for the positive and one for the negative versions) and connect variables that appear together in any clause. So if (a,b,c) were a clause, I would add 3 edges between a, b, c and another 3 edges between $\bar{a}, \bar{b}, \bar{c}$. A connection means that two vertices need to share the same value. After doing this, use DFS to ensure that a variable never connects to its opposite, and return True if it doesn't!

(b) One Of Each $3-\mathsf{SAT}$ is a variant where we are determining if there is an assignment where there is at least one true *AND* at least one false term per clause.
Hint: First consider One Of Each $4-\mathsf{SAT}$.

Solution: This one is NP complete. We first reduce $3-\mathsf{SAT}$ to One Of Each $4-\mathsf{SAT}$, where each clause has at most 4 terms and there must be at least one True and one False term in each clause.
Our strategy is to add an extra term to each $3-\mathsf{SAT}$ clause that we can assume to be false. Then solving One Of Each $4-\mathsf{SAT}$ is exactly equivalent to solving normal $3-\mathsf{SAT}$ on the original clauses.
The idea is simply to add the same dummy variable x_F to each clause of our 3-SAT instance to obtain a One Of Each $4-\mathsf{SAT}$ instance.
Here is why this reduction works.

- Say we had a satisfiable instance of $3-\mathsf{SAT}$. Our construction will have a "One of Each-satisfying" assignment if we set x_F to False.

- Say our construction has a "One of Each-satisfying" assignment. Here's the trick: if we flip the True and False values of every variable, we still have a "One of Each-satisfying" assignment for the formula! In one of these two cases, we'll have a "One of Each-satisfying" assignment that sets x_F to False. In that case, we know that at least one of every other term in each clause must be set to T. That assignment is a satisfying assignment for the original $3-\mathsf{SAT}$ formula.

Now we reduce $\mathsf{One\ Of\ Each}\ 4-\mathsf{SAT}$ to $\mathsf{One\ Of\ Each}\ 3-\mathsf{SAT}$ in the same way that we reduce 4SAT to 3SAT. For each clause with 3 or fewer terms, leave them alone. For each clause with 4 terms (a, b, c, d), create a dummy variable x and the two clauses $(a, b, x), (c, d, \bar{x})$. Here's the proof of correctness:

- Say there is a "One of Each-satisfying" assignment for the original 4-SAT instance. Then we have a few cases. Say there is a T and a F among a, b (or c, d). Then we can ignore the value of x in that clause, and set $x = c$ (or $\bar{x} = a$) to get a "One of Each-satsifying" assignment for our 3-SAT case. If there is a T and a F across the split, say $a = T, c = F$, then we can set $x = F$, and both clauses will have one of each True.
- Say there is a one of each True in both clauses in our 3-SAT construction. No matter what the value of x, there much be something among a, b that is the opposite of x, and something among c, d that is the opposite of $\bar{x}$. Thus among a, b, c, d, there must be at least one T and one F.

(c) $\mathsf{At\ Least\ Two}\ 3-\mathsf{SAT}$ is a variant where we are determining if there is an assignment where at least TWO terms per clause are True.

Solution: You can solve this in poly time. We reduce this to the problem of deciding if a bunch of logical implications are consistent or not, which we can solve in polynomial time using the strongly connected components algorithm (This was explained in a previous bonus problem).

A 2-element clause $(a \vee b)$ in $\mathsf{At\ Least\ Two}\ 3-\mathsf{SAT}$ means that both $a = True$ and $b = True$.

If you have 3 elements $(a \vee b \vee c)$ under $\mathsf{At\ Least\ Two}\ 3-\mathsf{SAT}$, then you can see that at least 2 of the above are true iff the formula $(a \vee b) and (b \vee c) and (a \vee c)$ is True. Each one of these terms $(a \vee b)$ is logically equivalent to $\bar{a} \to b$ (or $\bar{b} \to a$).

Thus given an instance of $\mathsf{At\ Least\ Two}\ 3-\mathsf{SAT}$, we can construct the graph corresponding to this set of logical implications, and decide if there is a valid satisfying assignment as was done in that previous bonus problem in polynomial time.