1: Winnie's Party

1. There is a polynomial time algorithm that can help Winnie invite the maximum number of popular, cool kids (i.e. people who will know at least 4 other people at the party). Our **algorithm** is as follows:

   (a) For each of the n potential guests, g_i, create a set, S_i, of all potential guests they know. For example, if g_1 knows g_2, g_3, and g_6 their set would be:

   $$S_1 = \{g_2, g_3, g_6\}$$

   (b) We will place all guests into list $I = \{g_1, g_2, ..., g_n\}$, marking each guest as a potential invite to the party. We will also create an empty list R, which will contain guests removed from I.

   (c) Next, we go through I and for each set S_i first remove any guests in S_i that overlap with guests in R (since these are no longer attending the party) and then check whether $|S_i| < 4$. If the set contains fewer than 4 guests, we add g_i to R and remove it from I.

   (d) We continue until either I is empty, or the contents of R remain unchanged over a full iteration of the remaining sets in I.

   Proof Let's now prove, via induction on the number of potential guests, n, that our algorithm works. In the base case, when $n \leq 4$ nobody can possibly know enough people to go. In this case, none of the guests will attend. Let's assume that for any $0 \leq i \leq n$ our algorithm correctly computes the maximum number of guests who'll attend the party. Let's now show that our algorithm works for $n + 1$ guests.

   Let's construct lists I and R as above, initially iterating through I from g_{n+1}. Doing a case analysis:

   (a) If g_{n+1} does not know at least 4 other potential guests, i.e. if $|S_{n+1}| < 4$, we remove g_{n+1} from I and place it in R. The size of I is now n, which by our inductive hypothesis we know we'll be able to correctly compute.

   (b) If $|S_{n+1}| \geq 4$, we iterate over the remaining guests in I. If any guest $\{g_1, g_2, ..., g_n\}$ knows fewer than 4 other guests, we'll remove it from I, decrementing I's size by at least 1. As in case (a), the size of I is now $\leq n$, which we know we can solve

for by our inductive hypothesis.

(c) If, however, we go through all guests in I and never change R, then according to our algorithm we're done — all guests in I know enough people to attend the party.

Our algorithm **only** excludes those guests who know too few people from attending the party, meaning if any guest knows ≥ 4 others attending, they'll attend. Thus, we maximize the party's attendees according to the problem's constraints.

Complexity All we need to do to prove this problem is not **NP**-hard is show that it runs in polynomial time. Given the size of $I = n$, in the worst case we will remove a single guest from I on each iteration. This means we'll iterate through the list:

$$\sum_1^n = 1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} = \mathrm{O}(n^2) \text{ times}$$

Each time we go through I from $1 \leq i \leq$ (size of I) we have to check for overlap between S_i and R. Even if we check for overlap linearly, in the worst case when $|S_i| = \frac{n}{2}$ and the size of $R = \frac{n}{2}$ this will be $\mathrm{O}(n^2)$. We do n^2 work n^2 times, therefore, this problem should be solvable in $\mathrm{O}(n^4)$, meaning it's not **NP**-hard.

It's important to note, even if we're somewhat off in our analysis such that our algorithm actually is $\mathrm{O}(n^3)$ or $\mathrm{O}(n^5)$ it doesn't really matter — both of these are polynomial run times meaning either is sufficient to prove this problem is not **NP**-hard.

2. Like above, there is a polynomial time algorithm to help Winnie invite the maximum number of *average* cool people i.e. guests who know at least 4 other guests while also not knowing at least 4 other guests. The algorithm is as follows:

   (a) As in *part 1*, create a set, S_i, for each potential guest, i, representing all the guests they know. Additionally, create a set for each potential guest of all the potential guests they don't know, T_i.

   (b) Now, run a slightly revised version of our algorithm from *part 1*. In step c of our algorithm above, we remove a guest from S_i if this guest is also in R. Now, remove this guest from either S_i or T_i if they exist in either set.

Proof We will prove correctness by induction on the number of potential guests, n. Our inductive hypothesis states that given n potential guests, we can find the largest

subset of these guests that both know *and* do not know at least 4 other guests in the subset. As a base case, if $n \leq 8$ we know no one can attend since everyone at the party needs to know at least 4 people and not know 4 people.

Now consider a potential guest list with $n + 1$ guests. There are 3 cases:

(a) If S_{n+1} or T_{n+1} represent a set with less than 4 other potential guests, we will remove them from I and place them in R. Now there are n guests in I and by our inductive hypothesis we know we will be able to correctly compute the maximum number of guests that will be invited to the party from the remaining n potential guests.

(b) If S_{n+1} or T_{n+1} represents a set with at least 4 other potential guests, we will iterate over the remaining guests in I. If any of them know fewer than 4 other guests, this will remove at least 1 guest from $|I|$, meaning we will only have $\leq n$ remaining potential guests, which we know we can solve for by our inductive hypothesis.

(c) If, however, we go through all guests in I and never change R, then we are done — all $n+1$ guests in I know enough people to attend the party and the algorithm is complete.

Our algorithm **only** excludes those guests from I that either do not know at least 4 people going to the party or know too many people (at least $n-4$ people attending the party). Therefore, after our algorithm completes, I contains the maximum number of guests that fit the criteria.

Complexity This algorithm has run time of $O(n^4)$ like the algorithm for *part 1*. This algorithm has the same mechanics as *part 1* except that it must compare R to both S_i and T_i. However, the total number of guests in S_i and T_i is $\leq n$ since these are mutually exclusive sets of guests. Therefore, the run time will still be $O(n^4)$.

Since we have proved that there is a polynomial time algorithm for this problem, we can say that it is not **NP**-hard.

3. We will prove that trying to network, i.e. inviting guests who know *at most* 4 other people at the party, is **NP**-hard via reduction. Let's take the independent set problem, which we know is **NP**-hard, and embed it within our "networking" problem. Thus, if we can solve our "networking" problem we'll also solve the independent set problem, showing our problem is at least **NP**-hard.

Consider an arbitrary instance, x, of the maximum independent set problem and a particular instance, y, of our networking problem. To translate x into y, we must add four nodes that attach to each node in the graph from x. These nodes only have one edge that connects them with one of the nodes from x.

Now, we will assume that we have a solution to y. We will show that x can only be solved if and only if y can be solved by showing that yes maps to yes and no maps to no between the two instances.

Assume that x has an arbitrary number of nodes n. This means that y has $5n$ nodes since we added 4 nodes to each node in x. We want to make a decision with both instances given some arbitrary k. We want to make sure that these decisions map between x and y. The decision question for x asks if there is a solution of size k. The decision question for y asks if there is a solution of size $4n + k$.

Proof Looking at our solution for y, we know that each of the additional $4n$ nodes are available to go to the party since they are only connected to one node and they can only know one other person going to the party (they are not particularly popular). We can always get a solution that is as good as the optimal solution of y for any $4n + k$ by sending each of the additional $4n$ nodes to the party and then choosing some independent set of original nodes to send to the party. To show this, consider two nodes each with their 4 additonal nodes going to the party. We can only have one of these two nodes go (total of 9 nodes). If both of them go, then we need to remove 1 additional node for each node (total of 8 nodes go). If we maximize the number of additional nodes going to the party, then we know that we are doing at least as well as the solution to y.

Yes Maps to Yes Given that we can find a solution to $4n + k$ using all $4n$ additional nodes, then we know that if we strip off these additional nodes, we have instance x with an independent set of size k. Further, if we have an independent set of size k in x, then we can have a solution to y of size $4n + k$ since we can choose this independent set of the original nodes and send all new additional $4n$ nodes to the party.

No Maps to No If we ask for a solution to y that invites $4n + k$ people to the party and no such solution exists, then there is also no independent set in x of size k. We know that this is true since a solution of y that is as good as optimal is choosing the $4n$ additional nodes and the k-sized independent set of the original nodes from x. If we cannot find an independent set in these nodes of y, then an independent set of size k in x also cannot exist. If we ask for a solution for x for an independent set of size k that cannot exist, then a solution to y of size $4n + k$ cannot exist, as explained above.

Given this decision solution, we can translate both of these to an optimization solution by asking for increasing large numbers of k until we can no longer find guest configurations.

The complexity of the translation from x to y is linear since we are adding a constant number of nodes for each node in x.

This reduction and the complexity of the translation proves that the solution to the networking problem is **NP**-hard.

2: Little League Coach

1. Given a coach who has offered to train n teams where any number of the teams' practices might overlap, we want to show that finding the maximum number of teams the coach can train without overlapping training times is **NP**-hard. Let's designate this the **little league** problem.

   Reduction We know that finding the maximum independent set of a graph is **NP**-hard. We will translate an independent set to our little league problem. Thus, if we can solve our problem we'll also have an independent set solution, ensuring our problem is at least **NP**-hard.

   Given an arbitrary independent set instance, x, let's construct a particular instance of our little league problem, y. Instance y will be a fully connected graph where, given a list of teams, L, where each team has a set of meeting times, the n nodes of our graph each represent teams while the edges between the nodes represent *any* overlap in their meeting times.

   Cost This is a polynomial time reduction. For each node in x we simply assign it a team in y. Thus, it's $O(n)$.

   Proof Let's assume we can solve this particular instance y. To show equivalence let's show that a valid solution to our little league problem maps to a valid independent set solution and vice versa:

   (a) **valid y maps to valid x**: A solution to y will pick teams such that none of the teams' meeting times overlap. Since in our graph we represent the conflict between any two teams as an edge, this means our solution to y won't ever choose two teams that share an edge; i.e. it'll never choose two nodes that are adjacent.

   Since our translation from x to y didn't require any additional "dummy" nodes, the solution to y, where no two chosen nodes are adjacent, maps directly to a valid independent set solution.

   (b) **valid x maps to valid y**: Likewise, given our arbitrary x, if we find a valid independent set then no two nodes selected will be adjacent. This maps directly to y — if no two nodes are adjacent, then no two nodes will have overlapping meeting times; making this a valid solution to y.

   Thus, y is at least as hard as x, making our little league problem **NP**-hard.

2. In this variation, we want to show that even if the coach's schedule can include at most one overlap, this is still **NP**-hard.

   Reduction & Cost: Similar to above, given an arbitrary independent set instance, x, we want to create a particular instance of our problem, y, where nodes represent teams and edges represent overlapping conflicts between the teams' meeting times. Furthermore, we want a y that is fully connected and where at least two teams have a *single* overlapping meeting time.

   After we construct our initial graph, let's additionally iterate through all teams and for each team with a *single* overlapping training session with some other team, c_i, connect it's node, n_i, to a dummy node, d_i. Let's then connect d_i to all other dummy nodes we might've created, in addition to all nodes connected to n_i (not including c_i).

   The dummy node effectively represents the connection between c_i & n_i, providing a viable, selectable node for this connection. d_i's connection to all other dummy nodes and all other nodes connected to n_1 prevents > 1 overlapping meeting times from being chosen.

   The cost of iterating through all nodes and finding those with a single overlapping meeting time is $O(n^2k^2)$. Additionally, we might have to construct up to n dummy nodes and connect them to almost every other node in the graph. This should have an additional cost of $O(n^2)$. Thus, we have a total cost of $O(n^2k^2 + n^2) = O(n^2k^2)$. Even if this running time is slightly off, it's clear that this is a polynomial time transformation.

   Proof Let's assume we have a solution to y. We'll prove equivalence by showing that for any j nodes chosen in x, it corresponds to $j + l$ nodes in y and vice versa, where:

   i. $\mathbf{l = 0}$ if we can find a solution to the problem without overlap (from 2.1) of size j but not a solution to the overlap problem of size $j + 1$
   ii. $\mathbf{l = 1}$ when there is a solution to both the non-overlap problem of size j and the overlap problem of size $j + 1$.

   (a) **j nodes in x maps to j + l nodes in y**: If the solution to x finds j nodes, we want to show that our solution to y finds $j + l$ nodes. In our original x, there is at least a single pair of nodes with a single edge between them, since this is the only way two teams can have a single overlapping meeting time. By definition, an independent set solution will only be able to choose a single node from this pair — it can't choose two adjacent nodes.

Our addition of dummy nodes to y, however, means that in the case where two nodes have a single overlapping meeting time, instead of only being able to choose one of the two, we can now select two nodes: one of the original nodes, (c_1 or n_i), and a node representing the connection between the two. This increases j by at most 1. Furthermore, the connection of our dummy node to all other dummy nodes and all nodes connected to n_i prevents j from increasing greater than $j+1$ — we'll be able to select a single, additional dummy node representing two nodes with a single overlapping meeting time, but the choice of this dummy node excludes selection of any additional dummy nodes or nodes adjacent to n_i. However, there is the case that adding a single overlapping meeting time will not yield more teams that the coach can work with and the solution from 2.1 will still be a maximal solution. Thus, for every j in x we get $j+l$ in y.

(b) **$\mathbf{j+l}$ nodes in y maps to j nodes in x**: Let's say our solution to y is able to find $j+l$ nodes. If the solution to y doesn't select any dummy nodes, then this is equivalent to finding $j+1$ nodes in x, meaning j nodes in x is a viable independent.

If our solution to y uses a dummy node, however, it's guaranteed via our construction to use **at most 1 dummy node**, since using more than one would correspond to having > 1 overlaps. We proved in 2.1 that if there is a solution to j in 2.1 (i.e. using 0 dummy nodes), then this will still map directly to x. Our original, arbitrary x didn't include any dummy nodes, thus if we parse these dummy nodes from y, we'll be left with only j selected nodes in x.

Thus, we've proven that j nodes in x corresponds to $j+l$ nodes in y, meaning if we're able to solve our instance of y, we'll get a corresponding solution to x with one less node. By solving this scheduling problem we're also solving an independent set problem, proving our problem is **NP**-hard.

3: Crosswords

We want to show that filling in a crossword is an **NP**-hard problem. We will do this via reduction, i.e. embedding within this crossword problem, another **NP**-hard problem. In this case, we'll choose set packing. Given an arbitrary set packing instance x, let's translate this into a particular instance, y, of our crossword problem.

For y, let's (1) define our dictionary of words to contain only m length words and n length words where $m \neq n$. Additionally, let's (2) stipulate that all words of length n consist of all A's and B's with at most one B. This implies that all viable m length words will consist of only A's and B's. And lastly, let's (3) restrict our crossword puzzles to $m \times n$ rectangles without any 'black squares'.

Reduction & Cost: So, let's say we have a puzzle and dictionary of words as described above. We want to translate the set packing problem into a variation of the crossword problem. In order to do this, for a set of elements S, let's map each element to a particular B position. The list L of sets in x, then, holds sets of elements each of which maps to a set of B positions for a particular m length word.

In order to create this, we need to go through each word of length m to determine the position of the B's in the word, so that we can map it correctly to a set in L. Given j words of length m, this encoding will take $O(jm)$. Thus, we can do this translation in polynomial time.

> Simply as a concrete example, given words "AABA" and "ABBB" in our dictionary, these might map to $\{b_3\}$ and $\{b_2, b_3, b_4\}$, respectively, in L.

Intuition: Constructing y as we have, what we notice is that because no n length words can have more than one B, we can't fit multiple m length words into our crossword with a B in the same position. For example, if we chose $\{b_2\}$ and $\{b_2, b_3\}$ to fill our crossword, this would mean an n length word would have ≥ 2 B's, which isn't viable in our dictionary. Thus, in order to fill our $m \times n$ crossword, we need to be able to choose at least n distinct, pairwise disjoint sets from L, which is exactly what set packing does.

Proof: Let's assume y is solved. We want to show that our crossword problem is at least **NP**-hard. In order to do this, let's show that if set packing can find k distinct, pairwise sets our crossword problem will be able to find k m length words to fill our crossword and vice versa:

1. **k in x implies k in y**: If set packing is able to find k pairwise disjoint sets in L, this means there is no overlap between any of the k sets' elements. As alluded to above,

since we're filling sets in L with the position of B's in the m length words, we can't select multiple words with the same B position since this would mean an n length word has ≥ 2 B's in it, which isn't viable.

Thus, all of the sets chosen in x will be viable selections in y, since none of the elements in the sets overlap. If set packing can find k pairwise disjoint sets our crossword will also be able to find k m length words to fill our crossword.

2. **k in y implies k in x**: If the solution to y is k, this means our crossword was able to find k m length words, none of whom have a B in an overlapping position. Thus, we'll be able to place all k of these words into our crossword without creating an *obvious* invalid word, i.e. an n length word with more than 1 B.

   Notice again that in order to do this y had to choose words such that there was no overlap among their B positions. Since we're packing sets with the B positions of the m length words, this is directly equivalent to the set packing problem: finding a particular number of sets where all the elements of the sets are pairwise disjoint (i.e. no two share the same element). If we're able to find k words to fill our crossword, we can also find k sets in the set packing problem.

Thus, we've shown that if we solve the crossword problem by finding k m length words to fit into the puzzle, we'll also have found k pairwise disjoint sets from list L. In order to solve our crossword problem, we also solve set packing with a polynomial translation, making this at least **NP**-hard.

Problem 4: Algorithm's Idol

For this problem, we need to find a polynomial 3-approximation to the optimal NP-hard algorithm. Our task is to find **cycles**: sequences where each unique person beats someone who beats another in the sequence and finished with the last person beating the first winner. We want to find these cycles and remove them within a 3-approximation of the optimal solution.

First, we will explain our approach to find which cycles we must remove. Every cycle must break down and contain a cycle of three contestants. We will prove via induction on the number of nodes n that for every cycle of n nodes there is a cycle of 3 nodes using three of the n nodes.

Our inductive hypothesis is that for every cycle of n nodes there is a cycle of 3 made of three of the n nodes. There are two base cases. The first is that there is a cyclical triangle of nodes, then we have found the a cycle of three. The second base case is that there is a cyclical quadrilateral (cycle of four nodes). Since there are four nodes in a cycle, any of the contests between contestants at opposite nodes will cause a cyclical triangle and we can return that triangle.

Now consider an arbitrary cycle with $n+1$ nodes. We will inspect the connections between every other node starting at an arbitrary node. Every other contestant must face each other since every contestant faces every other contestant. There are 2 cases here:

1. At least one of the connections between every other contestant are in the opposite direction of the $n+1$ cycle. If this is the case, then we will have a cyclical triangle between these two nodes and the middle node.
2. All connections are in the same direction as the $n+1$ contestants. In this case, we have a cycle with the $\lceil \frac{n+1}{2} \rceil$ contestants.
   (a) If $n+1$ is odd, then we arbitrarily look for a cycle using every other node until we would cross through the cycle that we are creating (after $\lfloor \frac{n+1}{2} \rceil$ nodes) and using the last connecting between the first node and the $\lfloor \frac{n+1}{2} \rceil$.
   (b) If $n+1$ is even, then we can take an arbitrary node in the cycle and look at the connections of every other node.

   With this new cycle of size $\lceil \frac{n+1}{2} \rceil$, we can find a cyclical triangular or quadrilateral according to our inductive hypothesis since $\lceil \frac{n+1}{2} \rceil < n$ for all $n > 1$ and if $n \leq 4$ then we have found a base case.

With either of these cases, we can find a cyclical triangle or quadrilateral which proves that every cycle breaks down to a cyclical triangle (or an inconsistent sequence of size 3).

Algorithm Our algorithm to remove the minimum number (within a 3-approximation) of contestants so that there are no inconsistent sequences of victories is as follows:

1. Create a directed graph of each of the contestants and their sing-offs where each node is a contestant and each edge is a sing-off. A victory is an edge pointing to another node and a defeat is an edge pointing to that node. For example A beats B so there is the direction of the edge between A and B is from A to B.

2. For each node in the graph, follow the victories. For each victory, look to see who these new nodes "beat". Now, we will look at each of these victories to see if any of these nodes beat the original node. We are only looking for inconsistent sequences of size 3 (cyclical triangles). For example, For instance, if with A beats B and B beats C and C beats A, we would start at A, then look at B, then look at C and notice that C points to A. If C does not point to A, then we will continue to loop through nodes.

3. As soon as we see an inconsistency, we remove all three of these nodes from the graph. and repeat step 2. We will continue to remove these triangular cycles until we have looped through each node.

The remaining nodes in the graph are consistent contestants and should remain in the competition.

Proof of Algorithm To prove that this algorithm returns the maximum number (within a 3-approximation) of contestants so that there are no inconsistent sequences, we will induct on the iterator of nodes (contestants) i.

We state our inductive hypothesis that of the i previously visited and remaining nodes in the graph, there are no triangular cycles. As a base case, if $i = 0$, then we have not visited any nodes and we have yet to remove any triangles.

Now consider the $i + 1$-th iteration through the n nodes. There are two cases that the algorithm will deal with:

1. This node is connected to two other nodes in a triangular cycle that needs to be removed. That is, this is an inconsistent sequence where A defeats B who defeats C who defeats A. Since we know from our inductive hypothesis that the remaining nodes visited by the previous i iterations are valid contestants and we know that this invalid contestant $i + 1$ has been removed from the graph, we can say that all remaining and previously visited nodes through the $i + 1$ iteration are valid.

2. This node is not a part of any triangular cycles. Therefore, there are no inconsistent sequences with who this contestant beats and who these contestants beat. This node $(i + 1)$ is valid and by the inductive hypothesis the remaining and previously visited i nodes are valid. Now we can say that the remaining and previously visited nodes through the first $i + 1$ iterations are valid and there are no inconsistent sequences of size three involving these nodes.

Given this conclusion, our algorithm will find and remove all inconsistent sequences of size 3. As proved above, all inconsistent sequences of an arbitrary size n contain an inconsistent sequence of size 3. Since there are no inconsistent sequences of size 3, there are no inconsistent sequences of any size in this graph.

This gives a 3-approximation of the maximum number of remaining contestants. The optimal solution would need to remove at least one of the contestants for each triangular cycle since without removing one of these, the triangular cycle will continue to exist. Further, since we remove the cycle as soon as we see it and move to the next node, we will never remove two cycles that contain the same node. For each contestant that the optimal solution would remove, our algorithm removes *at most* three. Therefore, our algorithm is a 3-approximation of the optimal solution.

Complexity The purpose of this approximation is to find a polynomial algorithm that can approximate an NP-hard solution. The first step of our algorithm is to create the graph. The graph has n nodes and each node has $n-1$ edges. Therefore, creating the graph is $O(n^2)$. The second and third steps of the algorithm iterate through each node (n nodes). At each node, we look at each of the node's victories and go to the node that they defeated (at most another $n-1$ nodes). At each of these nodes, we are looking at these victories (again at most $n-2$ nodes). Now, we must look through each of these victories to see if any of them point to the original node (another $n-2$ nodes). This part of the algorithm is therefore $O(n^4)$. Since $O(n^2)+O(n^4)=O(n^4)$, our algorithm runs in $O(n^4)$ time. This is polynomial.

Problem 5: Maximal Class Time

We will prove that trying to choose non-overlapping classes that meet twice is **NP**-hard via reduction. We will take the independent set problem of degree 3, which we know is **NP**-hard, and translate it to our class problem. Thus, if we can solve our class problem we will also solve the independent set problem of degree 3, showing our problem is at least **NP**-hard.

Consider an arbitrary instance, x, of the maximum independent set problem with a degree of at most 3 and a particular instance, y, of our class problem. To translate x into y, we must first create a set of paths through x where each node appears at most twice in the set of paths or twice in the same path. The greedy algorithm to define this set of paths is as follows:

1. Arbitrarily choose a node and arbitrarily traverse the graph until we have hit an endpoint.
2. If the original arbitrary node has an unused edge (one where there is no path), then we will start a path arbitrarily choosing connected edges that do not have a path. Once we hit an endpoint, we will merge the two endpoints at the original arbitrary node to create a longer path. If the new endpoint at the last visited node already has another endpoint from a different path, we will merge these two paths to create an even longer path.
3. We will continue steps 1 and 2 with a new arbitrary node that has unused edges until all edges have been covered in this set of paths.

We will prove that this greedy algorithm will cover each edge and give us a set of paths where each node appears at most twice (twice in the same path or once in two paths). Our algorithm will continue to create new paths until all edges are covered by going to the nodes that have an uncovered edge until no such nodes exist.

We will use a case analysis to show that no nodes will appear twice in the set of paths. Since a path cannot overlap previously used edges, we know that nodes that are of degree less than 3, must only appear at most twice in the set. A node of degree 1, has to be an endpoint since there is only one edge. A node of degree 2 cannot have more than 2 endpoints since we do not overlap edges. However, if the endpoints are ending two separate paths, we can merge these two endpoints to create one larger path. Thus, a node of degree 2 can belong to at most to one path and appear at most twice in the set.

If a node has a degree of 3, then there must be a path going through this node and an endpoint to another path or the same path. This is true in all cases of our algorithm. The only way that a node could appear thrice in our set of paths is if the node is the endpoint to three different paths since the node has a degree of 3. If there are two endpoints from

separate paths at a specific node, our algorithm will merge these to create a longer path leaving us with one path that goes through the node and an endpoint.

Now that we have this set of paths, we can assign times to each of the nodes with the constraint that connected nodes have a time conflict. The first time that a node appears in the set of paths, represents the first meeting of the class. If a class only appears once, it means that there are no conflicts with one of its meetings. If there is a second occurrence of the same node in the set, then this represents the second meeting of the class.

Now, we will assume that we have a solution to y. We will show that x can only be solved if and only if y can be solved by showing that yes maps to yes and no maps to no between the two instances.

Since conflicting classes are next to each other, every solution to y will only include classes that do not have a connected edge with another chosen class. More specifically, there is a solution to an independent set of degree three of size k if and only if there is a solution to our class problem where we can attend k classes without conflict.

We can prove this by looking at the decision version of these problems. If there are k classes that we can attend in our instance y, then we cannot select a node if its neighbor (conflicting class) is selected. This means that once we translate this solution back to x, we have an independent set of size k. If we cannot find a solution of size k to our instance y, this means that there are not k classes without conflict with one another. Further, we cannot have a solution to x in this case as well.

The same is true in reverse. If we can find an independent set in x, we can find a solution in y. If we find a solution in x, then there are no neighbors that are both selected and therefore there are no classes in y that have a conflict. If there is no independent set of size k, then we cannot find a set of classes of size k in y since there will be some conflict.

Given that we have reduced this problem to the NP-hard decision solutions, we can translate this to the NP-hard optimization problems. If we continue to ask if there exists a solution of size k where we increase k with each affirmative answer, we will eventually find the maximum number of classes that we can take.

The complexity of this translation must be polynomial to translate x into y. First, we need to analyze our greedy algorithm. Our algorithm greedily chooses nodes that have an unused edge. This means that we will go through each node *at most* 3 times since each node has a degree of 3. This algorithm is $O(n)$. We also must decorate each node with a class. This is also just visiting each node in the set of paths which as we proved is at most $2n$ or $O(n)$. Therefore, the translation is $O(n)$.

Since this NP-hard problem of an independent set of degree 3 can be reduced to our class problem in a polynomial translation, we can conclude that our class problem is also NP-hard.