1: Easy relatives of 3-SAT

(a) Truth table for the following problem may be written as:

x_1	x_2	$\neg x_1$	$x_1 \vee x_2$	$\neg x_1 \implies x_2$
0	0	1	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1

As we can observe in the truth table above, the output values of $x_1 \vee x_2$ and $\neg x_1 \implies x_2$ are same for respective values of x_1 and x_2, Hence both the expressions are logically same.

(b) Let us plot a graph between literals.

Assumption: We can assume that there exist a link from x_i to $\neg x_i$ and from $\neg x_i$ to x_i. Let us assume p to be a satisfying assignment value. So we can assume $p(x)$ = TRUE.
A new edge will be made in the graph if there is a clause $A \implies B$ or $\neg A \vee B$. This implies that if A is TRUE, B must be TRUE for the clause to be TRUE. If there is an connected edge between x_i to $\neg x_i$ and from $\neg x_i$ to x_i, the clause $A \implies B$ or $\neg A \vee B$ will give FALSE value for x_i equal to TRUE.
But this statement contradicts our assumption and hence we can say that it is not possible to have an edge connection between $\neg x_i$ and x_i. Similar case exists for $p(x)$ = FALSE.
Thus we can say that 2-CNF formula is satisfiable if and only if there is no variable x_i for which there is a path from x_i to $\neg x_i$ and vice versa.

Running time: Using Breadth-first search or Depth-first search we can find if x_i and $\neg x_i$ has an edge connection between them. This can be done in polynomial time. If an edge is found between them, we can discard as it wont be satisfiable.
Thus, 2-SAT has a polynomial time algorithm.

(c) We will assume a 3 - SAT instance as F = $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6)$. As we can observe in the first clause, $(x_1 \vee x_2 \vee x_3)$, according to the requirements, atleast 2 literals need to be TRUE to satisfy the instance. Thus we can say that the F instance is satisfied if and only if one literal in $(x_1 \vee x_2)$, $(x_2 \vee x_3)$, $(x_3 \vee x_1)$ is satisfied. For further clarification we can consider if x_1 is TRUE, $(x_1 \vee x_2)$, $(x_3 \vee x_1)$ is satisfied, and for one literal in $(x_2 \vee x_3)$ must be TRUE, x_2 or x_3 must be TRUE.
Thus we can see a similar case to 2 - SAT problem, atleast one literal in $(x_1 \vee x_2)$, $(x_2 \vee x_3)$, $(x_3 \vee x_1)$ should be TRUE. We know a polynomial time algorithm exist for a 2 - SAT problem. Hence we can conclude that 2-or-more 3 - SAT is in Polynomial time.

2: Decision vs Search

We can write the following pseudo code to show the explaination as required:
$Find\_Independent\_Set(G, k)$:
1. If FALSE is returned by Oracle O or else if k is 0, Stop and return FALSE.
2. Let $v \in G(V)$.
3. Let $Neigh(v)$ be the adjacent node of v.

4. We can further create 2 *subgraphs*, SG_1 and SG_2 with $G - v$ & $G - v - Neigh(v)$ respectively. Graph SG_2 is made as to make a graph such that if adjacent vertex are present for a particular vertex, it wont be having any independent set in it.
5. Now we can give SG_2 as input to Oracle O. If the given input value returns TRUE:
return $v \cup Find\_Independent\_Set(SG_2, k-1)$. Here as we see that one vertex is found, so we start recursing on $k - 1$.
6. Finally return $Find\_Independent\_Set(G, k)$

3: Reductions, Reductions

(a) Let's F be an instance of SAT, where
F = $(x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (x_5 \vee x_6 \vee x_7) \wedge (x_8 \vee x_9 \vee x_{10})$
As we know, for satifiablity of SAT problem, each clause should have atleast one literal as TRUE. For example: $(x_1 \vee x_2 \vee x_3 \vee x_4)$ must have atleast one literal as TRUE, which implies $(x_1 \vee x_2 \vee x_3 \vee x_4)$ is greater than or equal to one. Similarly, $(x_5 \vee x_6 \vee x_7)$ must have atleast one literal as TRUE, which implies $(x_5 \vee x_6 \vee x_7)$ is greater than or equal to one and so on.
This can be considered as an instance of Integer Linear Programming problem. The same proof can be extended for every clause in SAT instance, and thus Integer Linear Programming can be considered to be as hard as SAT problem.

(b) To Prove: Integer Linear Programming is an NP Complete Problem
As we see in the above problem - part (a), Integer Linear Programming is an NP hard problem. Thus if we prove that an instance of Integer Linear Programming can be verified in polynomial time, by giving an assignment, it can be proved that ILP is NP Complete. The above can be done by putting the corresponding assignment values into the instance comparing the inequality. This can be achieved in polynomial time. Thus, Integer Linear Programming is NP complete.

(c) LINEQ(modulo2) has a poly time algo.
Consider variable in 3-SAT be $z_1, z_2, z_3....z_n$. Thus we will have respective variables $y_1, y_2, y_3....y_n$ in ILP. We restrict each variable to be 0/1:
0 <= y_i <= 1 for all values to i.
Initializing y_i=1 in ILP and defining z_i as TRUE in the given formula, similar case for y_i=0 shows z_i=FALSE.
For every clause i.e. $(z_1 \vee \neg z_2 \vee z_3)$ will have limitations as:
$y_1 + (1 - y_2) + y_3 > 0$
To properly satisfy the above equation we should set $y_1 = 1$ or $y_2 = 0$ or $y_3 = 1$, which implies we must set z_1=TRUE or z_2=FALSE or z_3=TRUE in respective TRUTH assignments.

(d) Input Let there be f quadratic equations having boolean literals $x_1, x_2, ..., x_n$.
Here we can say that every equation is in following form:
$\Sigma_{i,j\epsilon[n]} x_i x_j = b$, where addition is for modulo 2.
if there exist a 0/1 assignment of variables that satisfy every equation then it is solvable.
To Show Quadeq is NP Complete
Proof If we find the decision problem belongs to NP - class for values of $x_1, x_2, ..., x_n$, we can verify that each equation is satisfied in poly time.
Lets proceed by reduction of 3- SAT to Quadratic Equation.
Reduction: We can reduce every clause in 3-SAT instance in an similar Quadratic Equation in

poly time using the following rules:
1. If a particular literal like x_i is in non-negative state then we can reduce it to simply k_i and if it exists in $\neg x_i$ state then we can reduce the literal to $(1 - k_i)$.
2. Multiply the reduced term with any variable such as p. Hence we can say that if x_i is TRUE in the clause, variable k_i will be having value 1 and vice-versa if x_i is FALSE then k_i will have a ZERO value.
3. Example: if any clause in 3-SAT instance is in the state $x_1 \vee \neg x_2 \vee x_3$ then its reduced form of Quadratic equation will be $pk_1 + q(1 - k_2) + rk_3 = 1$ modulo 2. Here p,q,r are boolean variables that can be modified to satisfy the requirement functions.
4. Hence, with respect to f clauses in 3-SAT, there will be f Quadratic Equations present.
Claim 1: If we consider 3-SAT instance as satisfiable then all Quadratic Equation will be satisfied.
We know that one literal in a 3-SAT must be TRUE, if a clause in 3-SAT instance is considered TRUE. Thus , this literal will give value 1 to the reduced state. The boolean variable attached with it will have value 1, rest all boolean variables can have value 0. Thus this equation will get equated to value 1. Same case is TRUE for satisfiability of all quadratic equation.
Claim 2: If we consider all Quadratic Equations as satisfied then 3-SAT instance is satisfiable.
We can observe that if a particular Quadratic Equation is equated to value 1, then atleast one reduced variable must have the value such that the respective literal in 3-SAT clause is TRUE, since Quadratic Equation can equate to one by changing the boolean variables, except when all terms multiplied are 0. Hence if all Quadratic Equations are satisfied then we can say that all the clauses will evaluate to TRUE.
Hence using claim 1 and 2 we can say that Quadratic Equations is NP Hard, and since it also belongs to NP- class, it must be NP complete.

4: Graphs - definitions

(a) We can prove the required statement using contradiction.

Assumption: Let G be a finite graph, that has no cycle. We assume that the graph G has longest path ending at vertex v_e
It is stated that Graph has degree greater than or equal to 2. As we have considered, since v_e is the last vertex of the longest path in Graph G, it must have a incoming edge. But according to statement "Graph has degree greater than or equal to 2" the other edge must join with some other vertex in the Graph, thus forming a Cycle. In other case vertex v_e will have to extend the longest path, which is contradicting the statement that Graph G has no cycle.
Thus Graph G must have a cycle if every vertex has degree greater than or equal to 2.

(b) As we know, in a Graph, the sum of degree of vertices is twice the number of edges.
Sum of degree of vertices in given graph is 35.
Thus accordingly the number of edges should be 17.5, that is not feasible.
Thus we can say that such a Graph is not possible.

(c) We can solve the following problem using Depth - first search. The previous or back edges can be detected. Counting of number of elements in stack happens when an edge is found.
1. Let a Graph be G. Mark all the nodes or vertices as 0 or unvisited.
2. Create an array N, with False or Null value for every vertex in Graph G.

3. Use depth - first search for every vertex v in Graph G.
4. If for a particular node, visited is FALSE or 0, then:
i. Mark this as 1 or visited, and change N for this node as TRUE.
ii. Now for next step, recurse on the neighboring vertices for this particular vertex. In case n is 1 or TRUE, for a vertex, we know that there must be a existing cycle.
iii. Next step is to find number or 1 or TRUE cases in N, if it is odd return all vertices that contains the 1 or TRUE value, else return FALSE, as there does not exist odd number of vertices.

5: Weary traveler

We can formulate the whole problem statement as:
Let:
Vertices be airports
Edges be flights with weights depart timing from origin airport as Depart_T and arrive timing at destination Airport be Arrive_T
We consider Time as a dictionary with key as node and value as the Arrival_T
Thus T[A] = Arrival_T at A start time.
Airport_Problem (Graph,A,InitialTime)
1. Consider T[A] with start time for the travel. Initially T[v] for all vertices that exist in the graph is considered as ∞
2. Priority Queue PQ exists of vertices on T. Here low T means high priority as main aim is to reduce time taken.
3. If PQ is not NULL, select a vertex v having the highest priority value.
4. For every vertex u in the adjacency list of v, create another Priority Queue, F for flights with Depart_T minus Arrive_T is greater than equal to 10 minutes, on Arrival_T where low Arrival time means higher priority. Thus in this way we can give priority to flights that arrive earliest.
5. Now, we find the flight with highest priority and let PT be its arrival time.
6. If for condition PT is less than equal to T[u], update T[u] with the values of ReturnT[B].

Running time: This is a modified version of Dijkstra's algorithm with Priority Queue as an additional feature.
Thus Dijkstra's algorithm takes $O(n^2)$ time
Creating priority queue will take $O(m)$ time in the worst case that is the number of flights.
So the total cost is $O(n^2) + O(m) = O(u^2)$.