1: Easy Relatives of 3-SAT

(a) Using a truth-table, we can prove that $x_1 \vee x_2$ and $\overline{x_1} \Rightarrow x_2$ are logically equivalent.

x_1	x_2	$\overline{x_1}$	$x_1 \vee x_2$	$\overline{x_1} \Rightarrow x_2$
0	0	1	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1

$\overline{x_1} \Rightarrow x_2$ can be restated as "if $\overline{x_1}$ is true, then x_2 must be true". On the first row of the truth table, we see a contradiction when $\overline{x_1}$ is true but x_2 is false. In all other cases the implication holds.

(b) 2-CNF formula is satisfiable $\iff$ there is no variable x_i for which there is a path from x_i to $\overline{x_i}$ and a path from $\overline{x_i}$ to x_i.

Proof LTR: 2-CNF formula is satisfiable $\Rightarrow$ there is no variable x_i for which there is a path from x_i to $\overline{x_i}$ and a path from $\overline{x_i}$ to x_i.

Given a path from x_i to $\overline{x_i}$ and a path from $\overline{x_i}$ to x_i we can view these paths as two conjunctions of implications.

The path $x_i \rightarrow x_j \rightarrow ... \rightarrow x_k \rightarrow \overline{x_i}$ and the path $\overline{x_i} \rightarrow x_k \rightarrow ... \rightarrow x_j \rightarrow x_i$ can be viewed as the following two conjunctions of implications.

$$(x_i \Rightarrow x_j) \wedge (x_j \Rightarrow ...) \wedge (... \Rightarrow x_k) \wedge (x_k \Rightarrow \overline{x_i})$$
$$(\overline{x_i} \Rightarrow x_k) \wedge (x_k \Rightarrow ...) \wedge (... \Rightarrow x_j) \wedge (x_j \Rightarrow x_i)$$

Given these two conjunctions of implications, there is no assignment of literals that will satisfy both equations.

Recall that T $\Rightarrow$ F is a false implication. In the final clause of both the above conjunctions, x_k and x_j will be true and either x_i or $\overline{x_i}$ will be false. Given a conjunction, if one clause is false the statement is false.

We have shown that if there exists a variable x_i such that there exists a path from x_i to $\overline{x_i}$ and a path from $\overline{x_i}$ to x_i, then the 2-CNF formula is unsatisfiable. Thus by contradiction, a 2-CNF formula is satisfiable only if there is no such variable x_i.

Proof RTL: 2-CNF formula is satisfiable $\Leftarrow$ there is no variable x_i for which there is a path from x_i to $\overline{x_i}$ and a path from $\overline{x_i}$ to x_i.

In proof LTR we viewed the paths from x_i to $\overline{x_i}$ and $\overline{x_i}$ to x_i as two conjunctions of implications. We showed given any assignment of literals x_i there does not exist a way to satisfy both conjunctions. We know the 2-CNF formula is satisfiable, meaning there does exist an assignment of literals x_i that satisfies all the paths it's graph representation contains.

Therefore, if the 2-CNF formula is satisfiable we know no such variable x_i exists.

Algorithm: Construct a directed graph with $2n$ vertices corresponding to x_i and $\overline{x_i}$ for $1 \leq i \leq n$. For each clause of the form $l_i \Rightarrow l_j$, place a directed edge from literal l_i to literal l_j.

For each pair of literals x_i and $\overline{x_i}$, we test if there exists a path from x_i to $\overline{x_i}$ and a path from $\overline{x_i}$ to x_i.

Starting at x_i, run BFS to determine if there exists a path to $\overline{x_i}$. If a path exists, starting at $\overline{x_i}$ run BFS a second time to determine if there exists a path to x_i. If both path exists, we are done and report that the 2-CNF formula used to construct the graph is unsatisfiable.

If all pairs of literals x_i and $\overline{x_i}$ are tested and the pairs of paths are not found, we report the 2-CNF formula is satisfiable.

Correctness: Using **Proof LTR** and **Proof RTL** we proved the statement: 2-CNF formula is satisfiable $\iff$ there is no variable x_i for which there is a path from x_i to $\overline{x_i}$ and a path from $\overline{x_i}$ to x_i.

Running time: In the worst-case (the 2-SAT formula is satisfiable and all pairs x_i and $\overline{x_i}$ must be considered) we will perform $2n$ BFS operations where n is the number of literals in the 2-SAT formula. The directed graph G representing the 2-SAT formula will contain n vertices and m edges where m is the number of clauses in the 2-SAT formula.

BFS has a time complexity of $O(n + m)$ and will be ran at most $2n$ times.

$\boxed{O(n(n + m))}$

(c) We can reduce the problem of *2-or-more-SAT* into 2-SAT.

Reduction: *2-or-more-SAT* $\leq$ *2-SAT*

For each clause in ϕ, we will construct a 2-CNF with three clauses that is logically equivalent. The conjunction of all the constructed 2-CNF clauses (to create one 2-CNF formula) will be logically equivalent to "2-or-more-satisfy" of the original 3-CNF clause.

Given a clause $(x_i \vee x_j \vee \overline{x_k})$ we will transform this into the 2-CNF $(x_i \vee x_j) \wedge (x_j \vee \overline{x_k}) \wedge (\overline{x_k} \vee x_i)$. This 2-CNF will be true if and only if 2-or-more of $x_i, x_j, \overline{x_k}$ are true. Meaning it is logically equivalent to "2-or-more-satisfy" of the original 3-CNF clause. Taking a conjunction of all of the three clause 2-CNF formulas will give one 2-CNF formula we will call $\phi\prime$.

Thus ϕ is *2-or-more 3-SAT* if and only if $\phi\prime$ is *2-SAT*.

By reduction, we have shown that *2-or-more 3-SAT* is as easy or easier than *2-SAT*. In part b we provided a polynomial time algorithm for *2-SAT*.

2: Decision vs. Search

Algorithm:

Setup: Create a set C and populate it with all $v \in V$ in G and a set S that is initially empty.

Step 1: If $\mathcal{O}(G, k)$ reports NO, report that an independent subset of size k does not exist in G. Otherwise continue.

Step 2: Randomly select and **remove** a vertex v from C. Remove v from G to form G'. Keep some state so we can add v back to G' to restore G.

Step 3: Call $\mathcal{O}(G', k)$ and if the oracle answers NO add v to S and add v back to G' to restore G. If the oracle responds with YES, replace G with G' (this vertex is discarded).

Step 4: If $|S| < k$ repeat from step 2. If $|S|$ equals k, S is an independent set of size k and we are done.

Correctness: From step 1, we know initially if G contains an independent subset of size k. If a vertex v can be removed from G to form G' and $\mathcal{O}(G', k)$ returns YES, we know G contains an independent subset of size k and v is not part of that subset. If $\mathcal{O}(G', k)$ returns NO, G no longer contains an independent subset so we know v is part of a independent subset so v is added to S.

We know the algorithm will finish because we only proceed from step 1 if an independent subset of size k exists. If one does, $|S|$ will equal k after at most $|V|$ iterations of the algorithm and $|V|$ is finite.

Running time: In the worst-case, each $v \in V$ in G must be removed from G (one at a time) and a call to $\mathcal{O}$ is made. Assuming the representation of the graph is an adjacency matrix, a node can be removed or added in $O(|V|)$ time. Giving $O(|V|)$ calls to $\mathcal{O}$ and $O(|V|^2)$ work to remove (and potentially add back) vertices.

3: Reductions, Reductions

(a) To prove the Integer Linear Programming (ILP) problem is NP-hard, we can provide a polynomial time reduction of 3-SAT (a known NP-complete problem) to ILP. 3-SAT is the problem of given a formula of clauses in conjunctive normal form (CNF) where each clause contains at most three literals.

Reduction: Let the variables in the 3-SAT problem be y_1, y_2, y_3, ..., y_n. There will be identical variables x_1, x_2, x_3, ..., x_n in our LIP problem restricted to the values $\{0, 1\}$ where 0 represents false and 1 represents true. Each linear constraint will contain the constants a_1, a_2, a_3, ..., a_n and b.

For each clause in the 3-SAT problem construct a linear constraint. If y_i is present then a_i is set to -1 otherwise a_i is set to 0. The constant b is set to -1. If y_i is negated in the clause, then x_i is set to $(1 - y_i)$ otherwise x_1 is set to y_i.

The formula is satisfiable if there exists integers x_i that satisfy all the linear constraints.

(b) The ILP problem is NP-Complete if it is in NP and is NP-Hard. By reducing a known NP-Complete problem (3-SAT) to ILP we showed ILP is NP-Hard. ILP is in NP because given integers x_i we can verify in polynomial time if they satisfy all the constraints. Thus ILP is NP-Complete.

(c) XOR-SAT is a concerned with determining if a formula containing clauses of XOR statements AND'ed together is satisfiable. LINEQ(mod2) can be reduced to XOR-SAT where each linear equation in LINEQ(mod2) is a clause in XOR-SAT. XOR-SAT can be solved using Gaussian elimination which has a $O(n^3)$ time complexity.

$\boxed{\text{YES}}$

(d) To prove that QUADEQ(mod 2) is NP-Complete we must show that it is NP-hard and that it is a member of NP. To show that it is NP-Hard we can reduce 3-SAT (a known NP-complete problem) to QUADEQ(mod 2). and we can show that it is in NP by showing it is verifiable in polynomial time.

Reduction: *3-SAT* $\leq$ *QUADEQ(mod 2)*

Given a 3-CNF formula ϕ, we can form a series of quadratic equations such that they satisfy QUADEQ(mod 2) if and only if ϕ is 3-SAT.

For each clause in ϕ we must form a quadratic equation mod 2 that is logically equivalent. A quadratic equation mod 2 can be expressed in algebraic normal form (ANF). An ANF formula is one or more terms ANDed together and each clause is XORed together. For quadratic, each clause must contain exactly two literals.

We convert each clause in ϕ of the form $(x_i \vee x_j \vee x_k)$ to ANF.

$$(x_i \wedge x_j) \ \oplus (x_i \wedge x_k) \ \oplus (x_j \wedge x_k) \ \oplus (x_i \wedge x_j \wedge x_k) \ \oplus x_i \ \oplus x_j \ \oplus x_k$$

Each clause must contain exactly two literals. A clause with one literal is trivial to make two literals, as x_i and $(x_i \wedge x_i)$ are logically equivalent. With three literals, we must create a new literal of the form x_{ij}. To give x_{ij} the same value as $(x_i \wedge x_j)$ we include the quadratic equation $x_{ij}^2 + x_i x_j = 0$ in our series of equations. This enforces $x_{ij} = 1$ if and only if $x_i x_j = 1$.

$$(x_i \wedge x_j) \ \oplus (x_i \wedge x_k) \ \oplus (x_j \wedge x_k) \ \oplus (x_{ij} \wedge x_k) \ \oplus (x_i \wedge x_i) \ \oplus (x_j \wedge x_j) \ \oplus (x_k \wedge x_k)$$

Each equation has the form $\sum_{i,j} c_{ij} x_i x_j = b$ and there will be an equation for each clause in ϕ (plus the equations that enforce the values of the new literals). b is set to 1 for all equations. The relevant to the clause c_{ij} are set to 1 while the rest are set to 0.

The relevant c_{ij} for each equation are the c that contain of the combinations of literals present in the clause the equation is representing. The c relevant to the example clause are:
$c_{ij},\ c_{ik},\ c_{jk},\ c_{ijk},\ c_{ii},\ c_{jj},\ and\ c_{kk}$. We are only concerned with the literals that are contained in the clause and c can be seen as a flag to consider only the relevant part of the quadratic equation for the clause. Negated literals such as $\overline{x_i}$ are assigned a value (1 - x_i) naturally.

The formula ϕ is satisfiable by some assignment of variables x_i if the constructed quadratic equations are satisfied by the same assignment of variables x_i. Proving we have reduced 3-SAT to QUADEQ(mod 2) showing that QUADEQ(mod 2) is NP-hard.

Exists in NP: To show that QUADEQ(mod 2) is in NP, we must show in polynomial time that given a specific assignment of variables x_i QUADEQ(mod 2) is satisfied by them. This is trivial to do as if we have variables x_i, we can simply plug in the variables and do arithmetic on a polynomial number of quadratic equations.

NP-Complete: We have shown QUADEQ(mod 2) is NP-Hard and is in NP thus it is NP-Complete.

4: Graphs - Definitions

(a) A connected and undirected graph G has a cycle if all of its vertices have degree ≥ 2.

Proof: The *degree sum formula* states, given an undirected graph G, the sum of the degrees of all the vertices V in G is equal to twice the number of edges E in G. Where the degree of a vertex is the number of edge incident to the vertex.

$$\sum_{v \in V} deg(v) = 2|E|$$

Given a graph with n vertices each with degree ≥ 2

$$\sum_{v \in V} deg(v) = 2|E| \geq 2n$$
$$|E| \geq n$$

When all vertices have a degree ≥ 2, there are at least n edges in the G.

Properties of trees: A tree is defined as a connected acyclic graph. A tree has the maximum number of edges an acyclic graph can contain. Given a tree with n vertices there are exactly $n - 1$ edges in the tree. If any edge is added to a tree, a simple path is formed.

We have shown that G has at least n edges. Using this along with the properties of trees, we have proven that G must have a cycle.

(b) There cannot exist an undirected graph consisting of 10 nodes with degrees 2, 3, 4, 4, 7, 1, 4, 5, 3, 2 respectively.

Proof: We consider the sum of the given vertices.

$$\sum_{v \in V} deg(v) = 2|E|$$
$$35 = 2|E|$$
$$|E| = 17.5$$

The number of edges must be an integer value. Thus we know that the given graph does not satisfy the *degree sum formula* and is an invalid graph.

(c)

Algorithm: Run breadth-first search on G. Using the colors black and white, we will color the vertex as we traverse G. For each vertex, assign it the opposite color of it's parent's color (if it's parent is white - assign it back and vice-versa). If we arrive at an already colored vertex and it's parent is the same color, we have found a cycle of odd length that contains these two nodes. If the search completes and this situation doesn't arise, G does not contain any negative cycles and we can report such.

If a cycle of odd length does exist, we can find it by running a second breadth-first search on G, keeping track of the previous vertex for each vertex. Say vertex v is the child of vertex u and they have the same color. Start the second breadth-first search at vertex v and proceed until vertex u is found. The cycle is the path $u \rightarrow v$ (created using the previous references) and the edge(u, v).

Correctness: Using properties of bipartite graphs, we prove if G contains a cycle of odd length.

A graph G is bipartite if and only if it does not contain an odd cycle.

A graph is bipartite if and only if it is 2-colorable.

We can see a graph G is non-bipartite if it contains a cycle of odd length. To show G is non-bipartite we show that it is not 2-colorable (a parent and child vertex are assigned the same color). If G is 2-colorable, we know it is bipartite and doesn't contain any cycles of odd length.

Running time: Given a graph G, we run breadth-first search at most two times on G. G contains $|V|$ vertices and $|E|$ edges. In breadth-first search we enqueue and dequeue $O(|V|)$ vertices and these operations have a time complexity of $O(1)$. All edges are explored which is $O(|E|)$. Therefore the total complexity of two breadth-first searches is $O(|V| + |E|)$.

$\boxed{O(|V| + |E|)}$

5: Weary Traveler

Algorithm: A directed graph G is constructed where airports are represented by vertices and flights are represented by edges. Each vertex contains the name of an airport. Edges contain the arrival time of the flight, departure time of the flight, and a flight id that uniquely identifies the flight (used to construct flight schedule when the shortest travel path is found). If there are multiple flights between two airports, the flights are combined into one edge and each flight is checked. G is represented with an adjacency list, where each vertex has a list of it's neighbors. Given a list of flights, this is easy to construct.

Four dictionaries are used to keep track of the following: the previous vertex in a path, the minimum travel time from the source to a given vertex, what time you will arrive at a given vertex to achieve the minimum travel time to that vertex from the source, and an id of the flight used to arrive at the a given vertex.

The helper method *arrival(u, w)* returns the arrival time of the edge that connects u and w, *depart(u, w)* returns the departure time, and *id(u, w)* returns the flight id.

Algorithm 1 Minimum Total Travel Time

```
1:  Initialize
2:      travel_time[v] = ∞ For all v ∈ V in G
3:      prev[v] = null For all v ∈ V in G
4:      arrival_time[v] = −∞ For all v ∈ V in G
5:      flight_id[v] = null For all v ∈ V in G
6:
7:      travel_time[source] = 0
8:      arrival_time[source] = time arrived at source airport
9:      PriorityQueue q = [source]
10: end Initialize
11:
12: procedure MINIMUM_TOTAL_TRAVEL_TIME(G, SOURCE, DEST)
13:     while q is non-empty do
14:         u = q.peek() # Get the top priority item but don't remove it
15:         if u == dest then
16:             return schedule with smallest total travel time
17:         end if
18:         for neighbors w in u do # Consider each flight in each vertex with an edge from u
19:             for flight in edge(u, w) do
20:                 new_travel_time = arrival(u, w) - arrival_time[u] + travel_time[u]
21:                 if arrival_time[u] + 10 ≤ depart(u, w) && new_dist < travel_time[w] then
22:                     travel_time[w] = new_travel_time
23:                     arrival_time[w] = arrival(u, w)
24:                     flight_id[w] = id(u, w)
25:
26:                     q.update(w) # Add w to the priority queue or update it's priority
27:                 end if
28:             end for
29:         end for
30:         q.pop() # Remove from the queue the vertex with the highest priority
31:     end while
32:     return No path exists
33: end procedure
```

The algorithm is Dijkstra's algorithm with small modifications. On line 14, if we de-queue the destination vertex we are done and can construct the shortest path. This is accomplished using the *prev* and *flight_id* dictionaries starting at the destination vertex and working backwards to the source vertex. On line 19, the travel time to travel from u to w is calculated, combining flight time and time spent waiting in airports. In contrast to traditional weighted graphs, this is necessary since time will be spent traveling while inside airports (nodes) and while on flights (edges).

A priority queue implemented with a min-heap is used to store the vertices waiting to be processed so we can efficiently get the vertex with the highest priority. The travel time from the source to the vertex is stored in the priority queue with the vertex and is used to determine priority.

Correctness: We proved the correctness of Dijkstra's algorithm in class.

Running time: A graph is constructed containing n vertices and m edges. Assuming the reputation of the graph is an adjacency list, the construction will have a time complexity of O(n + m).

In Dijkstra's algorithm the worst-case is when the destination vertex is the last vertex dequeued and the while loop runs for each $v \in V$. For each iteration of the while loop, $deg(u)$ neighbors are considered. Assuming the queue is a priority queue implemented as a min-heap, update and pop have a $O(\log n)$ time complexity. All other operations (such as calculating arrival times and peeking at the top of the queue) have a time complexity of $O(1)$. Dijkstra's runs in linear time with respect to the size of the graph with $O(\log n)$ queue operations. Thus the overall running time is $O((n + m)\log n)$. This dominates the time complexity of the construction of the graph and is our overall time complexity.

$\boxed{O((m + n)\log n)}$

Collaboration and Sources

Question 1: collaboration with Maks Cegielski-Johnson

Question 2: collaboration with Maks Cegielski-Johnson

Question 3: collaboration with Maks Cegielski-Johnson

Question 3c: Wikipedia - Boolean satisfiability problem

Question 4a: Wikipedia - Tree (graph theory)

Question 4c: Wikipedia - Bipartite graph