**Q1**

**Theorem.** *3-*$\textsc{Col} \leq_p$ *3-*$\textsc{Sat}$.

*Proof.* Given any string $w$, we want to define a polynomial time computable function $f$ such that $f(w) \in$ 3-$\textsc{Sat}$ iff $w \in$ 3-$\textsc{Col}$. Since we have a reasonable encoding of undirected graphs, we can detect if $w$ encodes an undirected graph in polynomial time and output a garbage string $\neg\neg$ if $w$ isn't a graph. Now, assume $w = \langle G \rangle$ is an undirected graph.

The variables of $f(w)$ are $\{u_C : u \in V(G), C \in \{R, G, B\}\}$. The idea is colouring a vertex $u$ to green corresponds to assignments $u_G = \textsc{True}$, $u_R = \textsc{False}$, $u_B = \textsc{False}$. Same goes for red and blue. We know $G$ is 3-colourable iff no edge contains the same colour at both its vertices for some colour assignment. This inspires the following reduction.

For each edge $uv$, construct a 3-$\textsc{Cnf}$ formula $good_{uv}$ describing $uv$ has different colours at both ends. The initial expression is derived by observing that $uv$ is good exactly when $u$ is red and $v$ is not red, or $u$ is green and $v$ is not green, or $u$ is blue and $v$ is not blue:

$$\begin{aligned}
good_{uv} \equiv &(u_R \wedge \neg v_R) \vee (u_G \wedge \neg v_G) \vee (u_B \wedge \neg v_B) \\
\equiv &[(u_R \vee u_G) \wedge (u_R \vee \neg v_G) \wedge (\neg v_R \vee u_G) \wedge (\neg v_R \vee \neg v_G)] \vee (u_B \wedge \neg v_B) \\
\equiv &(u_R \vee u_G \vee u_B) \wedge (u_R \vee \neg v_G \vee u_B) \wedge (\neg v_R \vee u_G \vee u_B) \wedge (\neg v_R \vee \neg v_G \vee u_B) \\
&\wedge (u_R \vee u_G \vee \neg v_B) \wedge (u_R \vee \neg v_G \vee \neg v_B) \wedge (\neg v_R \vee u_G \vee \neg v_B) \\
&\wedge (\neg v_R \vee \neg v_G \vee \neg v_B).
\end{aligned}$$

The last expression above is what we will write for $good_{uv}$.

One more thing to ensure is that the colour assignment is valid, so every vertex is exactly one colour. For each vertex $v$, construct a 3-$\textsc{Cnf}$ $good_v$:

$$\begin{aligned}
good_v \equiv &\neg(v_R \wedge v_G \wedge v_B) \\
&\wedge \neg(v_R \wedge v_G \wedge \neg v_B) \\
&\wedge \neg(v_R \wedge \neg v_G \wedge v_B) \\
&\wedge \neg(\neg v_R \wedge v_G \wedge v_B) \\
&\wedge \neg(\neg v_R \wedge \neg v_G \wedge \neg v_B) \\
\equiv &(\neg v_R \vee \neg v_G \vee \neg v_B) \\
&\wedge (\neg v_R \vee \neg v_G \vee v_B) \\
&\wedge (\neg v_R \vee v_G \vee \neg v_B) \\
&\wedge (v_R \vee \neg v_G \vee \neg v_B) \\
&\wedge (v_R \vee v_G \vee v_B),
\end{aligned}$$

which says not all three colours are assigned to $v$, no two colours are assigned to $v$, and at least one colour is assigned to $v$. The second expression will be in $f(w)$ as $good_v$.

Finally, we output

$$f(w) = \left( \bigcup_{uv \in E(G)} good_{uv} \right) \wedge \left( \bigcup_{u \in V(G)} good_{u} \right).$$

We will show that $f$ is indeed a polynomial time reduction.

This is clearly a polynomial time algorithm, since we only need to iterate once through all the edges and vertices of $G$ to output $f(w)$ ($good_v$ and $good_{uv}$ are both constant size).

Now suppose $\langle G \rangle \in$ 3-Col. Then, we can find a colour assignment such that each edge connects vertices of different colours. For each vertex $u$, if $u$ is coloured green, then set $u_G =$ True and $u_B = u_R =$ False. Same for other colours. This is clearly a satisfying assignment: for any edge $uv$, $good_{uv}$ is satisfied because each edge connects vertices of different colours; for any vertex $v$, $good_v$ is satisfied because each vertex has exactly one colour assigned. Thus, $f(w)$ has a satisfying assignment and so is in 3-Sat.

Conversely, suppose $w$ is such that $f(w) \in$ 3-Sat. Then, $f(w)$ has a satisfying truth assignment $\phi$. Colour $G$ by the rule $v$ is coloured $C$ iff $\phi(v_C) =$ True. This is indeed a colouring of the vertices since all of $good_v$'s are satisfied (exactly one of $u_R, u_G, u_B$ is true for any $u$), so our rule is unambiguous. Finally, no two vertices of an edge $uv$ have the same colour because $good_{uv}$ is satisfied.

Thus, 3-Col $\leq_p$ 3-Sat via $f$. □

## Q2

**Theorem.** TriangleFree *is $\mathcal{NP}$-Complete.*

*Proof.* We show this by reduction from a known $\mathcal{NP}$-complete problem IndependentSet.

Similar to Q1, since we have reasonable encoding method, the garbage cases where input $w$ is not of the form $\langle G, k \rangle$ can be handled in polynomial time. Assume $w = \langle G, k \rangle$, we will construct in polynomial time graph $G'$ and natural number $k'$ such that $\langle G, k \rangle \in$ IndependentSet if and only if $\langle G', k' \rangle \in$ TriangleFree.

The key of the construction is to "translate" edges into triangles. Formally, the set vertices of $G'$ is $V(G') = V \cup V'$, where $V$ is the vertices of $G$ and $V'$ is the set of duplicates of $V$-vertices, denoted using the prime notation (each $v \in V$ corresponds to $v' \in V'$). The edges of $G'$ is $E(G') = E \cup E' \cup C$, where $E$ is the edges of $G$ and for each $uv \in E$, there are corresponding edges $uv', u'v \in E'$. The edges $C$ connects each vertex with its duplicate. For convenience throughout the proof, unprimed notation denotes vertices in $V$, primed notation denotes vertices in $V'$. Intuitively, any vertex $u$ and its duplicate $u'$ are indiscernible to the vertices in $V$: they have the exact same neighbours. Finally, let $k' = |V| + k$.

This algorithm is clearly polynomial time, since it requires iterating through once to duplicate vertices and edges. And $k'$ of course can be computed easily. The extra edges between original $V$-vertices and duplicates are also doable within one iteration.

Now, we will show that this is indeed a reduction.

Suppose $\langle G, k\rangle$ is such that $G$ has an independent set of at least $k$ vertices. We show that $G'$ has a triangle-free subset of size at least $k'$. Since $G$ has an independent set $I$ of at least $k$ vertices, we can assume WLOG it has exactly $k$ vertices. This is because removing vertices from an independent set still results in an independent set. We claim that $V' \cup I$ is a triangle-free set of size $k' = |V| + k$. Without a doubt, its size is $k'$ because $V'$ is disjoint from $I$ ($V'$ are duplicates and $I$ are originals), and $|V'| = |V|$, $|I| = k$. Moreover, there are no triangles in $V' \cup I$. There are no edges between elements of $V'$ because edges from duplicates only connect them with original $V$-vertices. Thus, any triangle must contain two vertices in $I$. This is not possible, because $I$ is an independent set, so there cannot be an edge connecting any two of its elements. Thus, $V' \cup I$ is a triangle-free set of size $k'$. This shows that $G'$ has a triangle-free set of size at least $k'$.

Conversely, suppose that $G$ has no independent set of size at least $k$. This in particular means there is no independent set of size $k$. We show that $G'$ has no triangle-free subset of size at least $k'$. It suffices to show that there is no triangle-free subset of size exactly $k'$, since adding vertices to a subset with triangle doesn't remove the triangle. Let $S \subseteq V(G')$ with size $k'$. We will use a variant of the pigeonhole principle: each original vertex $v \in V$ has a bucket, and elements of $S$ are added to the buckets ($v$ and $v'$ are added to the $v$-bucket). Notice that each bucket contains at most two pigeons. There are $|V|$ buckets and $|V| + k$ pigeons, thus, there are $k$ buckets that contain both a vertex and its duplicate. In the graph, this means $S$ contains $k$ pairs $v_1, v_1', \ldots, v_k, v_k'$[1]. However, we know that $G$ has no independent set of size $k$. Thus, for some $1 \leq i < j \leq k$, $v_i v_j \in E$. By construction, $v_i v_j' \in E'$ and $v_j v_j' \in C$. This makes a triangle $v_i - v_j - v_j'$. And we are done. □

**Q3** a) The meaning of $A \in \text{co}\mathcal{NP}$-complete is taken to be: $A \in \text{co}\mathcal{NP}$ and everything in $\text{co}\mathcal{NP}$ reduces to $A$ in polytime.

**Theorem.** Taut *is* co$\mathcal{NP}$*-complete.*

**Lemma.** *$A$ is co$\mathcal{NP}$-complete if and only if $A^c$ is $\mathcal{NP}$-complete (this says co($\mathcal{NP}$-complete) = co$\mathcal{NP}$-complete).*

*Proof.* ($\longrightarrow$) Suppose $A$ is co$\mathcal{NP}$-complete. Then $A$ is in co$\mathcal{NP}$, so $A^c$ is in $\mathcal{NP}$. And

$$\begin{aligned} & B \in \mathcal{NP} \\ \Longrightarrow & B^c \in \text{co}\mathcal{NP} \\ \Longrightarrow & B^c \leq_p A \\ \Longrightarrow & B \leq_p A^c. \end{aligned}$$

This shows $A^c \in \mathcal{NP}$-complete.

[1] Minor detail: we can assume $k > 1$ because every one-vertex set is independent.

($\longleftarrow$) Suppose $A^c$ is $\mathcal{NP}$-complete. Then $A^c$ is $\mathcal{NP}$, so $A$ is co$\mathcal{NP}$.

$$\begin{aligned} &B \in \text{co}\mathcal{NP} \\ \Longrightarrow &B^c \in \mathcal{NP} \\ \Longrightarrow &B^c \leq_p A^c \\ \Longrightarrow &B \leq_p A. \end{aligned}$$

Thus, $A$ is co$\mathcal{NP}$-complete. □

We will now prove the main theorem.

*Proof.* Define Faut, the set of formulas that are false under any assignment. Note that Faut $=_p$ Taut: they both reduce to each other via the map $w \mapsto \neg w$. The reduction works because for any propositional formula $\phi$, $\phi$ is false under any assignment if and only if $\neg\phi$ is true under any assignment. The garbage strings are also automatically handled because $\neg w$ is still garbage if $w$ is. Now, we show Faut is co$\mathcal{NP}$-complete. By the lemma, it is enough to show Faut$^c \in$ $\mathcal{NP}$-complete.

Faut$^c \in \mathcal{NP}$ is easy to see via a verifier. Given $\langle w, c\rangle$, if $w$ is not a propositional formula, we can detect it and return false in polynomial time (because of reasonable encoding); otherwise, the certificate $c$ should be a variable assignment that satisfies $w$. Evaluating an assignment is polytime.

We show that Faut$^c$ is $\mathcal{NP}$-hard by reduction from Sat. Given any string $w$: if $w = \phi$ is a formula, output $w$ as is; otherwise, output $x \wedge \neg x$. This is clearly polynomial time. If $w = \phi$ is satisfiable, then $\phi \in$ Faut$^c$ as well by definition of Faut. Conversely, if the output $w'$ is in Faut$^c$, it must be that the input is $w'$ (because $x \wedge \neg x$ isn't in Faut$^c$). But the formulas in Faut$^c$ are all satisfiable. Thus, this a valid polytime reduction. So Faut$^c$ is $\mathcal{NP}$-hard.

This shows Faut$^c$ is $\mathcal{NP}$-complete; hence Faut is co$\mathcal{NP}$-complete. By transitivity of $\leq_p$ and Faut $\leq_p$ Taut, we know anything in co$\mathcal{NP}$ reduces to Taut. Also, because Taut $\leq_p$ Faut, we have Taut$^c \leq_p$ Faut$^c$. And by $\mathcal{NP}$-completeness of Faut$^c$, we can conclude Taut $\in$ co$\mathcal{NP}$.

Therefore, Taut is co$\mathcal{NP}$-complete. □

b)

**Theorem.** Taut *is* $\mathcal{NP}$ *implies* $\mathcal{NP}$ *equals co*$\mathcal{NP}$.

*Proof.* (co$\mathcal{NP} \subseteq \mathcal{NP}$) If $A \in$ co$\mathcal{NP}$, then $A \leq_q$ Taut because Taut is co$\mathcal{NP}$-complete. But Taut $\in \mathcal{NP}$, so $A \in \mathcal{NP}$.

($\mathcal{NP} \subseteq$ co$\mathcal{NP}$) If $A \in \mathcal{NP}$, then $A^c \in$ co$\mathcal{NP}$. So $A^c \leq_p$ Taut because Taut is complete. But Taut is also $\mathcal{NP}$, so $A$ is co$\mathcal{NP}$. □

**Q4** a) Consider the following algorithm for Factor with an oracle for Div.

```
Factor(x)
1  Intialize x' = x, factors = [ ]
2  while ⟨x', x' − 1⟩ ∈ Div
3      Initialize l = 2, r = x' − 1
4      while r ≠ l
5          m = ⌊(l + r)/2⌋
6          if ⟨x', m⟩ ∉ Div
7              l = m + 1
8          else r = m
9      Update x' = x'/l and append l to factors
10 Append x' to factors
11 Return factors
```

The idea is to find the prime factors one by one using binary search while reducing $x'$ by the prime factor found. If $x'$ is not a prime[2], an iteration of the outer loop (line 2) is run. Each iteration of the outer loop finds the least prime divisor of $x'$ using the inner loop (line 4). We will prove the correctness and running time using the following loop invariant.

**Loop invariant:** on the $k$th iteration of the inner loop, the least prime factor of $x'$ is in $[l_k, r_k]$ (subscript denotes the value of variables after the $k$th iteration); moreover, $r_k - l_k \leq \lfloor (x'-3)2^{-k} \rfloor$.

*Proof.* We proceed by induction. Before the first iteration ($k = 0$), $l_0 = 2$ and $r_0 = x' - 1$. Indeed, the prime factor is in $[2, x'-1]$ because $x'$ is not a prime. Also, $r_0 - l_0 = x' - 3 = (x'-3)2^{-0}$.

Suppose this works for some $k$ and the loops runs for one more iteration, we show the $k+1$ case holds. There are two cases depending on the condition on line 6. If $\langle x', m \rangle \notin \textsc{Div}$, then this means the smallest prime factor is not in $[l_k, m]$, so it must be in $[m+1, r_k]$ (by induction hypothesis the factor is in $[l_k, r_k]$). Thus, updating $l_{k+1} = m + 1$ is correct. Now,

$$\begin{aligned} r_{k+1} - l_{k+1} &= l_k - \lfloor (l_k + r_k)/2 \rfloor - 1 \\ &\leq l_k - \frac{l_k + r_k}{2} + \frac{1}{2} - 1 \quad \text{because } \frac{l_k + r_k}{2} - \lfloor \frac{l_k + r_k}{2} \rfloor \leq \frac{1}{2} \\ &= \frac{r_k - l_k}{2} - 1/2 \\ &\leq \lfloor (x'-3)2^{-k-1} \rfloor. \end{aligned}$$

Similarly, if $\langle x', m \rangle \in \textsc{Div}$, then the smallest prime divisor is in $[l_k, m]$. So

[2]For $x' \geq 2$, $x'$ is prime iff $x'$ has no divisor other than 1 and $x'$ iff $\langle x', x'-1 \rangle \notin \textsc{Div}$

updating $r_{k+1} = m$ is correct. And,

$$\begin{aligned} r_{k+1} - l_{k+1} &= \lfloor (l_k + r_k)/2 \rfloor - l_k \\ &= \lfloor (l_k + r_k - 2l_k)/2 \rfloor \\ &\le \lfloor (r_k - l_k)/2 \rfloor \\ &= \lfloor (\lfloor (x' - 3)2^{-k} \rfloor)/2 \rfloor \\ &\le \lfloor (x' - 3)2^{-k-1} \rfloor. \end{aligned}$$

□

*Proof of Termination, Correctness, and Running Time.* Now, it is clear that the algorithm terminates. Given any $x'$, the inner loop terminates as soon as $2^k > (x' - 3)$. And the outer loop terminates because every time $x'$ loses one prime factor, and eventually $x'$ will be a prime, which would terminate the loop.

It is also clear that the algorithm terminates with the correct factorization. The algorithm is correct on prime input $x$: the outer loop is never entered and it directly outputs a list containing $x$ itself. On composite inputs $x = x'$, the loop invariant guarantees that when $r = l$, the smallest prime factor of $x'$ is $l$. So every iteration of the outer loop adds one new prime factor $l$ to *factors*, and removes $l$ from the factorization of $x'$. It is well-known that the prime factors of $x'$ is $l$ combined with the prime factors of $x'/l$ (with multiplicity). And by induction on the number of prime factors, our algorithm produces the correct factorization (base case is $x$ is prime, which is covered above).

Finally, the running time of this algorithm is $O((\log x)^k)$ for some $k$. The inner loop runs at most $\log x' \le \log x$ steps by the loop invariant. The outer loop runs at most $\log x$ steps because $x$ contains at most $\log x$ number of prime factors (2 is the smallest prime factor, and $x$ can have at most $\log x$ many of them). Each inner loop iteration takes $O(\log x)$ time because addition and division by 2 is cheap. Similarly, the operations on line 9 are doable in $O((\log x)^{k'})$ for some $k'$, because multiplication is polynomial time and so is copying $\log x$ bits). Combining with the fact that both loops run at most $\log x$ times, we can conclude the overall running time is $O((\log x)^k)$ for some $k$.

This shows FACTOR $\xrightarrow{p}$ DIV. □

b)

**Theorem.** DIV $\in \mathcal{NP} \cap co\mathcal{NP}$.

*Proof.* DIV $\in \mathcal{NP}$ can be shown by making a polynomial time verifier. The certificate for $\langle x, y \rangle$ is simply a divisor $d$ of $x$ satisfying $1 < d \le y$. The verifier simply checks if $1 < d \le y$ and $d | x$. As division/modulo and $<$ can all be performed in polynomial time, the verifier is polynomial time.

DIV $\in$ co$\mathcal{NP}$ is also shown by making a polynomial time verifier. The certificate for $\langle x, y \rangle$ is a prime factorization $p_1, p_2, \ldots p_k$, for $x$. If $k > \log x$, or if any prime is greater than $x$, return false. Then, the verifier checks if each $p_i$ is a prime using the AKS primality test and $p_1 \dots p_k = x$. If not, the verifier returns false. If they are all prime, the verifier iterates through to see if there is a $p_i$ such that $p_i \leq y$. The verifier returns true if and only if there is such a $p_i$[3]. This runs in polynomial time: AKS is invoked at most $\log x$ times, each time taking $O(\log x)$; checking $p_1 \dots p_k = x$ involves multiplying at most $\log x$ numbers each with at most $\log x$ bits; $<$ comparisons can be done in $O(\log x)$. Therefore the verifier is polynomial time. $\square$

[3] $x$ has a divisor $d$ with $1 < d \leq y$ if and only if $x$ has a prime divisor $p$ with $1 < p \leq y$.