Q1

Theorem. *If $\mathcal{NP} = \mathcal{P}$, then $EXP = NEXP$.*

Proof. It is clear that $EXP \subseteq NEXP$, since any deterministic TM that runs on $O(2^{n^k})$ is also an $O(2^{n^k})$ nondeterministic TM. Thus, it remains to prove that $NEXP \subseteq EXP$.

Let $A \in NTIME(2^{n^k})$ (assume $k \geq 1$, otherwise this is constant time and is trivial). We show that A can be decided by a deterministic TM in $O(2^{n^{k'}})$ for some $k' \in \mathbb{N}$. We do this by first showing a similar language A' is in $\mathcal{NP}$, and then use the fact that $\mathcal{NP} = \mathcal{P}$ to obtain a good deterministic algorithm.

Since $A \in NTIME(2^{n^k})$, we can find an NTM N that decides A in $O(2^{n^k})$. Consider the language $A' = \{x\#^{2^{|x|^k}} | x \in A\}$ (if # is part of alphabet, replace with something that is not in the alphabet). A polynomial time NTM decider N' can be constructed for A' as follows: on input w, read the input tape from left to right once and reject if the input is not of the form $x\#^{2^{|x|^k}}$ (done by counting the number of symbols before #); then, run N on the portion that does not contain #; finally, accept or reject according to N's output. Notice the first step of checking is extremely crucial, because if we don't check for that, we might run N on the entire input string, which is not polynomial time.

The algorithm above is correct because it accepts a string w if and only if w is of the form $x\#^{2^{|x|^k}}$ and $x \in A$, which is what A' is. This is polynomial time because checking the string takes at most polynomial time, and running N on x takes $O(2^{|x|^k})$ time, which is linear in the input length $|x| + 2^{|x|^k}$.

Thus, we have proven $A' \in \mathcal{NP}$. By $\mathcal{NP} = \mathcal{P}$, we get that $A' \in \mathcal{P}$ via a polynomial time TM M'. Suppose M' halts in n^l steps for large enough inputs and some $l \in \mathbb{N}$. The final step is to show that $A \in EXP$. Consider a TM M that decides A in exponential time: on input w, append $\#^{2^{|w|^k}}$ to w; run M' and output what M' outputs.

The machine M is correct because it accepts w if and only if $w\#^{2^{|w|^k}} \in A'$. For the running time: appending $\#^{2^{|w|^k}}$ takes $|w| + 2^{|w|^k}$ steps (traverse to the end and append); and running M' takes at most $\left[|w| + 2^{|w|^k}\right]^l$ steps. Combining, it runs for at most

$$\begin{aligned} |w| + 2^{|w|^k} + \left[|w| + 2^{|w|^k}\right]^l &\leq \left[|w| + 2^{|w|^k}\right]^{l+1} \\ &\leq \left[2^{|w|^{k+1}}\right]^{l+1} \\ &= 2^{(l+1)|w|^{(k+1)}} \\ &\leq 2^{|w|^{k+2}} \end{aligned}$$

steps, for w long enough. This shows $A \in EXP$, and we are done. □

Q2 An $\langle m, n, k\rangle$-game is tic-tac-toe but with an $m \times n$ board (player X goes first). Define $GT = \{\langle m, n, k\rangle | \text{Player X has a winning strategy}\}$.

Theorem. *$GT \in PSPACE$.*

Proof. We shall solve this recursively via the function HasStrat(C, P, k), which takes an $m \times n$ board configuration C with at least one unfilled cell, and $P \in \{\text{X}, \text{O}\}$ is the player that goes first; HasStrat(C, P, k) tells us if P has a winning strategy on C:

```
HasStrat(C, P, k)
1  for (i, j) such that C[i][j] is not filled
2       copy C' = C
3       set C'[i][j] = P
4       return TRUE if Win(C', P, k) returns TRUE
5       if C' is completely filled, return FALSE
6       call HasStrat on C' and the other player
7       if HasStrat returns FALSE
8            return TRUE
9  return FALSE
```

The helper function Win(C, P, k) checks if C is a winning configuration for P, and it is implemented as follows:

```
1  for (i, j) such that 1 ≤ i ≤ m, 1 ≤ j ≤ n
2       if i + k − 1 ≤ m and C[i][j], ..., C[i + k − 1][j] = P
3            return TRUE
4       if i + k − 1 ≤ m and j + k − 1 ≤ n and C[i][j], ..., C[i + k − 1][j + k − 1] = P
5            return TRUE
6       if i + k − 1 ≤ m and j + k − 1 ≤ n and C[i + k − 1][j], ..., C[i][j + k − 1] = P
7            return TRUE
8       if j + k − 1 ≤ n and C[i][j], ..., C[i][j + k − 1] = P
9            return TRUE
```

The subroutine Win is clearly correct. If there are k of P's in a row or column, it will be detected by the first and last if clauses; if it is diagonal, it will be detected by the second or third if clauses.

For each empty cell $C[i][j]$, the algorithm HasStrat checks if putting P in that cell leads to P winning. If so, it returns true because that is a winning strategy for P on configuration C. If that does not win the game for P, the algorithm recurses to see if the other player has a winning strategy if we put P at $C[i][j]$; if the other player has no winning strategy, the algorithm returns true because this means P has a winning strategy. Finally, if none of the above worked, P unfortunately does not have a winning strategy and false is returned.

Then, a decider M for GT simply runs HasStrat(C_0, X, k), where C_0 is the empty configuration.

There are two things we need to show. The first thing is that M is correct, and the second is that $M \in PSPACE$.

M is correct: Note that M is correct as long as HasStrat is correct. Termination is easy to see because every time the algorithm recurses, the number of empty cells decreases by 1; and no more recursion happens when there is 1 empty cell. Correctness can be proved by induction on the number of spaces left in C.

Base case is when there is exactly one empty cell in C. The algorithm puts P in that cell, and checks if that yields a winning configuration for P. If it does, the algorithm returns true, which is correct. If it does not yield a winning strategy, the algorithm returns false on the next line because the configuration is full. This is also correct. Thus, our base case is correct.

Suppose HasStrat is correct for any configuration that has $k \in \mathbb{Z}^+$ empty cells, we show that HasStrat is correct for any configuration that has $k+1$ empty cells. Suppose HasStrat is called on C, P with $k+1$ empty cells. We show the return value is true if and only if P has a strategy. As usual, if the algorithm returns true on line 4, it is because filling $C[i][j]$ with P gives a win for P, so indeed P has a winning strategy. If the algorithm returns true on line 8, this means if P is put at $C[i][j]$, the other player P' has no winning strategy (induction hypothesis guarantees this is correct). If P' has no winning strategy (i.e winning strategy for P' means there is a move P' can make such that it is possible for P' to win no matter P's next moves are), that would mean: for any move P' makes, there are moves P can make to make it impossible for P' to win; this just means a winning strategy for P. So the algorithm's return value of true is correct. If the algorithm returns false, it must do so on line 9 because $k \in \mathbb{Z}^+$. Returning false on line 9 means no matter where we put P, the other player P' has a winning strategy, so P does not have a winning strategy. This is correct as well. This completes the induction step.

Thus, HasStrat is correct, and so is M.

M is in $PSPACE$. Note that the subroutine Win uses $\log m + \log n + \log k$ space to store i, j, k. So clearly it is $PSPACE$. Also, the recursion depth for HasStrat(C_0, X, k) is at most mn since at each recursion, the number of empty cells decreases. At each depth, the recursion stack stores C, taking mn space, and uses $\log m + \log n + \log k$ space to store i, j, k. By reusing space, we can conclude that in total the amount of space used is $O((mn)^2 + \log k)$.

Combining everything above, we have shown HasStrat is in $PSPACE$. □

Q3 Given a 2-CNF formula ϕ, associate a directed graph G_ϕ where vertices are variables of ϕ and their negations and edges each clause $l_1 \vee l_2$ gives rise to $(\neg l_1, l_2)$ and $(\neg l_2, l_1)$.

Theorem. *2-Sat is NL-complete.*

Proof. a) Suppose l_1 and l_2 are two nodes in G_ϕ such that there is a path

$$l_1 = v_0 \to \ldots \to l_2 = v_k.$$

We know by construction of G_ϕ that $v_i \to v_{i+1}$ comes from the clause $\neg v_i \vee v_{i+1}$,

which also gave rise to the edge $\neg v_{i+1} \to \neg v_i$. Thus, we also get a path

$$\neg l_2 = \neg v_k \to \ldots \to \neg l_1 = \neg v_0.$$

Moreover, if σ is a truth assignment that satisfies ϕ and l_1. Let i be such that $\sigma(v_i) = T$. Then, since $\neg v_i \vee v_{i+1}$ is a clause, $\sigma(\neg v_i) = F$, and $\sigma(\neg v_i \vee v_{i+1}) = T$, we must have that $\sigma(v_{i+1}) = T$. This says $\sigma(v_i) = T \implies \sigma(v_{i+1}) = T$. By repeated application, we can see $\sigma(l_2) = T$.

b) We show that ϕ is unsatisfiable if and only if G_ϕ has a directed cycle including both x and $\neg x$ for some variable x.

($\longleftarrow$) Let there be a cycle in G_ϕ be a cycle containing variable x and $\neg x$. We show ϕ is unsatisfiable. Suppose by contradiction that σ is a truth assignment such that $\sigma(\phi) = T$. In the first case, $\sigma(x) = T$. Then since we have a cycle containing x and $\neg x$, we get a path from x to $\neg x$. By a), this means $\sigma(\neg x) = T$ as well, a contradiction. In the second case, $\sigma(\neg x) = T$. Since we have a cycle containing x and $\neg x$, we get a path from $\neg x$ to x. By a), $\sigma(\neg x) = T$ implies $\sigma(x) = T$, a contradiction. Thus, σ cannot exist and ϕ is unsatisfiable.

($\longrightarrow$) Suppose there is no cycle that contains a variable and its negation in G_ϕ. We give a procedure to find a satisfying assignment σ of ϕ. Suppose $\phi = (u_0 \vee u_1) \wedge \ldots \wedge (u_{2k} \vee u_{2k+1})$. Our procedure aims to satisfy ϕ clause by clause from left to right. For $i = 0, \ldots, k$:

- If neither of $\sigma(u_{2i})$ nor $\sigma(u_{2i+1})$ has been given a value, then
  - If there is a path in G_ϕ from u_{2i} to $\neg u_{2i}$, set $\sigma(\neg u_{2i}) = T$ and propagate T in G_ϕ from $\neg u_{2i}$ (i.e set any node that $\neg u_{2i}$ has a path to to T).
  - Otherwise, set $\sigma(u_{2i}) = T$ and propagate T in G_ϕ from u_{2i}.

  Explanation: Notice our assignment ensures that if u has a path to v in G_ϕ and $\sigma(u) = T$, then $\sigma(v) = T$. The propagation does not contradict previous assignments: if u has a path to v and v has been assigned false, then u would already be false because T propagated from $\neg v$ to $\neg u$ when $\sigma(v)$ was set. The propagation is consistent with our new assignment for u_{2i} because G_ϕ does not contain cycles with u_{2i} and $\neg u_{2i}$ (for example, if we set $\sigma(u_{2i}) = T$, then T does not propagate to $\neg u_{2i}$ because there is no path from u_{2i} to $\neg u_{2i}$; similarly for when we set $\sigma(\neg u_{2i}) = T$).

  Note that this satisfies the clause $(u_{2i} \vee u_{2i+1})$. In the first case, we set $\sigma(\neg u_{2i}) = T$; notice that $\neg u_{2i} \to u_{2i+1}$ is an edge, so $\sigma(u_{2i+1}) = T$ by propagation. In the second case, $\sigma(u_{2i}) = T$. Thus, the ith clause is satisfied in either case.

- If u_{2i} or u_{2i+1} has been set to T or F, continue.

  Explanation: This satisfies the ith clause for the same reason as given above.

Finally, set the remaining unassigned variables to T. Their values do not matter because ϕ has been satisfied.

This shows that ϕ is satisfiable and we are done.

c) We show 2-SAT is NL-complete by showing 2-SAT^c is NL-complete. Since $NL = coNL$, 2-SAT$^c \in NL$ implies that 2-SAT $\in NL$. Furthermore, if $B \in NL$, then $B^c \in NL$, $B^c \leq_L$ 2-SAT^c, $B \leq_L$ 2-SAT. Therefore we would have 2-SAT be NL-complete.

An important observation is that ϕ is a reasonable encoding of the graph G_ϕ. Note that PATH $\in NL$ with a reasonable graph encoding, we shall use this as a subroutine in our specification.

2-SAT $\in NL$: Consider the following NTM N that decides 2-SAT^c. On input w:

If w is not of the form $(u_0 \vee u_1) \wedge \ldots \wedge (u_{2k} \vee u_{2k+1})$, return true. Otherwise, call the input ϕ. For $i = 0, \ldots, 2k+1$, if PATH$(G_\phi, u_i, \neg u_i)$ and PATH$(G_\phi, \neg u_i, u_i)$ return true, then return true. Return false after all $2k$ literals are tested.

This is clearly correct by our result in b): ϕ is unsatisfiable if and only if some variable x and its negation are in a cycle (i.e that there is a path from x to its negation and from the negation to x). This only uses log space because storing i, u_i, $\neg u_i$ take log space. The calls to PATH takes space logarithmic in the number of nodes, which is less than the length of ϕ. So the calls to PATH take at most log space. Thus, 2-SAT$^c \in NL$.

2-SAT is NL-hard: We reduce from PATH using a log space transducer T. Given a graph G and nodes s and t, T outputs formula ϕ_G

$$\left(\bigwedge_{(u,v) \in E(G)} (\neg u \vee v) \right) \wedge (s \vee s) \wedge (\neg t \vee \neg t).$$

This can clearly be printed using log space because it only involves iterating over all the edges in G. We now show that G has a path from s to t if and only if ϕ_G is unsatisfiable.

($\longrightarrow$) Suppose there is a path $s = v_0 \to \ldots \to t = v_k$ and, for a contradiction, that σ is a satisfying assignment. Because of the clause $(s \vee s)$, we know $\sigma(s) = T$. And similarly $\sigma(t) = F$. But then each clause $(\neg v_i \vee v_{i+1})$ forces $\sigma(v_{i+1}) = T$. Repeatedly apply this, we see that $\sigma(t) = T$, a contradiction.

($\longleftarrow$) Suppose ϕ_G is unsatisfiable, we show that G has a path from s to t.

Consider the graph G_{ϕ_G}—this graph has G as a subgraph as well as a subgraph "opposite" to G. In more details, G is a subgraph of G_{ϕ_G}: every node of G is a literal of ϕ_G, which is a node of G_{ϕ_G}; similarly, every edge of G is in G_{ϕ_G}. Moreover, there is a subgraph $\neg G$ of G_{ϕ_G} where the nodes are the negations of the nodes in G, and the arrows are reversed. For example, $u \to v$ in G implies $(\neg u \vee v)$ is in ϕ_G, which means $\neg v \to \neg u$ is in G_{ϕ_G}. Finally, because of $(\neg t \vee \neg t)$

in ϕ_G, $\neg G$ connects to G via $t \to \neg t$ and nothing else[1]. Similarly, $\neg s \to s$ is the only edge from $\neg G$ to G.

Now, by b), we get a cycle containing x and $\neg x$ for some variable x. However, because G and $\neg G$ are connected only by $t \to \neg t$ and $\neg s \to s$, we get a path

$$\underbrace{x \to \ldots \to t}_{\text{in } G} \to \underbrace{\neg t \to \ldots \to \neg x \to \ldots \neg s}_{\text{in } \neg G} \to \underbrace{s \to \ldots \to x}_{\text{in } G}.$$

Observe that we can concatenate $s \to \ldots \to x$ with $x \to \ldots \to t$ to get a path from s to t.

Thus, we have PATH $\leq_L$ 2-SAT^c. Hence 2-SAT^c is NL-complete, and so is 2-SAT.

□

[1] nothing else because other edges are within G or $\neg G$