1. Solution:

We can convert a standard deterministic, one-tape TM $M_1$ to an equivalent deterministic, two-tape, read only input TM $M_2$ as follows. TM $M_2$ first copies its input on tape 1 to tape 2, then simulates $M_1$ on tape 2. These two TMs use exactly the same space because we consider only the cells that are used on tape 2 when measuring $M_2$’s space complexity.

Conversely, we can convert a deterministic, two-tape, read only input TM $M_2$ to an equivalent standard deterministic, one-tape TM $M_1$ using the standard simulation of two-tape TMs by one-tape TMs in the proof of Th 3.13. Then if $M_2$ uses $O(f(n))$ space, $M_1$ uses $n + O(f(n))$ space because on $M_2$ we don’t count the input tape, but on $M_1$ we do count the $n$ cells that are used to simulate that tape. We are assuming that $f(n) \geq n$, so $n + O(f(n))$ is $O(f(n))$ and both TMs have the same asymptotic space complexity.

2. 8.10 The Japanese game go-moku is played by two players, “X” and “O” on a $19 \times 19$ grid. Players take turns placing markers, and the first player to achieve five of her markers consecutively in a row, column, or diagonal is the winner. Consider this game generalized to an $n \times n$ board. Let $GM = \{\langle B \rangle \mid B$ is a position in generalized go-moku, where player “X” has a winning strategy\}.

By a position we mean a board with markers placed on it, such as may occur in the middle of a play of the game, together with an indication of which player moves next. Show that $GM \in \text{PSPACE}$.

Solution:

The following algorithm decides whether player “X” has a winning strategy in instances of go-moku; in other words, it decides $GM$. We then show that this algorithm runs in polynomial space. Assume that the position $P$ indicates which player is the next to move.

$M =$ “On input $\langle P \rangle$, where $P$ is a position in generalized go-moku:

1. If “X” is next to move and can win in this turn, $accept$
2. If “O” is next to move and can win in this turn, $reject$
3. If “X” is next to move, but cannot win in this turn, then for each free grid position $p$, recursively call $M$ on $\langle P' \rangle$ where $P'$ is the updated position $P$ with player “X”’s marker on position $p$ and “O” is next to move. If one or more of these calls accepts, then $accept$. If none of these calls, then $reject$.
4. If “O” is next to move, but cannot win in this turn, then for each free grid position $p$, recursively call $M$ on $\langle P' \rangle$ where $P'$ is the updated position $P$ with player “O”’s marker on position $p$ and “X” is next to move. If all of these calls accept, then $accept$. If one or more of these calls rejects, then $reject$.”

The only space required by the algorithm is for storing the recursion stack. Each level of the recursion adds a single position to the stack using at most $O(n)$ space and there are at most $n^2$ levels. Hence, the algorithm runs in space $O(n^3)$.

3. Solution:
We first prove that a graph is bipartite iff it has no odd cycles. Assume to the contrary that G is bipartite, but \( C = (x_0, x_1, \ldots, x_{2k}) \) is an odd cycle in G on \( 2k + 1 \) vertices for some \( k \geq 1 \). Then the vertices \( \{x_1, x_3, \ldots, x_{2k-1}\} \) must lie on one side of the partition and the vertices \( \{x_0, x_1, \ldots, x_{2k}\} \) on the other side of the partition. But \( (x_0, x_{2k}) \) is an edge in G between two vertices in the same side of the partition, contradiction.

Conversely, assume G has no odd cycles. Work with each connected component independently. Let v be an arbitrary vertex of G. Then, because G has no odd cycles, placing all vertices of G which are an even distance from v on one side and those which are an odd distance from v on the other side gives a bipartition of G.

Here is an NL algorithm for the language BIPARTITE = \{⟨G⟩ : G is a bipartite graph \}. The algorithm is similar to the NL algorithm for PATH. It nondeterministically guesses a start node s and then nondeterministically guesses the steps of a path from s to s (i.e. a cycle containing s). It uses a counter to keep track of the number of steps it has taken and rejects if this counter exceeds the number of nodes in G. If the algorithm reaches s after an odd number of steps then it accepts.

The algorithm uses \( O(\log n) \) space. It needs to store only the counter, the start node s, and the current node of the path, each of which requires \( \log n \) bits of storage. Thus BIPARTITE ∈ NL, and so BIPARTITE ∈ coNL. But NL=coNL and hence BIPARTITE ∈ NL.

4. Recall that a directed graph is strongly connected if every two nodes are connected by a directed path in each direction. Let STRONGLY-CONNECTED=\{⟨G⟩ : G is a strongly connected graph \}. Show that STRONGLY-CONNECTED is NL-complete.

Solution:

To see that STRONGLY-CONNECTED is in NL is we loop over all pairs of vertices and nondeterministically guess a connecting path for each pair of vertices. We accept if we find a path for each pair.

To show that STRONGLY-CONNECTED is NL-complete, we reduce from PATH. Given ⟨G, a, b⟩ (assume directed graph G has more than 2 nodes) build graph \( G' \) as follows:

Begin with G. For every node x not equal to a, add the edge \((x, a)\). For every node y not equal to b, add the edge \((b, y)\). Now we prove this is indeed a logspace mapping reduction.

If G has a directed \( a \rightarrow b \) path then \( G' \) is strongly connected since for each pair of nodes \((x, y)\), we can get from x to y by the edge \((x, a)\), then an \( a \rightarrow b \) path and finally the edge \((b, y)\).

Conversely, if there is no \( a \rightarrow b \) path in G then there is no \( a \rightarrow b \) path in \( G' \) (we cannot use the new edges \((b, y)\) because we cannot get to b and the edges \((x, a)\) are useless because we can already reach a), and so \( G' \) will not be strongly connected.

Notice that our reduction loops over all pairs of nodes and so uses only logspace to determine what pair of nodes we are currently working with.