1. (a) Consider the following algorithm:

Algorithm 1 Is-Cycle-Limited-Version(G)

Require: $G = (V, E)$ is a directed graph with n vertices such that $\forall u \in V\ d_+(u) \leq 1$.
Ensure: Returns *true* if G contains a cycle and $false$ otherwise.

```
for i ← 1 to n do
    u ← V[i]
    k ← 0
    while ∃v ∈ V such that (u, v) ∈ E do
        u ← v {there can be only one such v since d+(u) ≤ 1}
        k ← k + 1
        if k = n then
            return true
return false
```

Space Analysis:
At any given time, we only need to keep $i, k, u, v \Rightarrow O(\log n)$.

Correctness:
($\Rightarrow$) Assume G contains a cycle C. When Is-Cycle-Limited-Version's for-loop reaches i such that $V[i] \in C$, the while condition will always be *true* since we always pick $v \in C$. Thus, after n steps, we get $k = n$ and the algorithm will return *true*.

($\Leftarrow$) Assume Is-Cycle-Limited-Version returned *true* during the ith iteration. Then, it has found a directed path P of length $n+1$ in G, starting at $V[i]$. Since G only contains n vertices, P is not simple and thus contains a cycle.

(b) Consider the following algorithm:

Algorithm 2 Is-Cycle(G)

Require: $G = (V, E)$ is a directed graph with n vertices.
Ensure: Returns *true* if G contains a cycle and $false$ otherwise.

```
guess i ∈ {1, ..., n}
u ← V[i]
for k ← 1 to n + 1 do
    guess j ∈ {1, ..., n}
    v ← V[j]
    if (u, v) ∈ E then
        u ← v
    else
        return false
return true
```

Space Analysis:
$O(\log n)$ for $i, j, k, u, v \Rightarrow O(\log n)$.

Correctness:
($\Rightarrow$) Assume G contains a cycle C. Then, some execution path will begin in some $u \in C$,

follow edges along C, complete the for-loop and return *true*. Thus, Is-Cycle returns *true*.

($\Leftarrow$) Assume Is-Cycle returned *true*. Then, there is a path of length $n+1$ in G. As $|V| = n$, this path cannot be simple, thus contains a cycle.

2. Let $B \in$ co-C. By definition, $\overline{B} \in C$. A is C-complete, so there exists a (polynomial, logarithmic, ...) reduction f from $\overline{B}$ to A, mapping $x \in \overline{B}$ to $f(x) \in A$ and $x \notin \overline{B}$ to $f(x) \notin A$. In other words, f maps $x \notin B$ to $f(x) \notin \overline{A}$ and $x \in B$ to $f(x) \in \overline{A}$. But this makes f a (polynomial, logarithmic, ...) reduction from B to $\overline{A}$. Hence, $\overline{A}$ is co-C-hard.

   $A \in C$, thus $\overline{A} \in$ co-C; Therefore $\overline{A}$ is co-C-complete.

3. (s,t)-Con is NL-complete, so by Question 3, $\overline{\text{(s,t)-Con}}$ is co-NL-complete. If we had $\overline{\text{(s,t)-Con}} \in NL$, that would mean $NL \subseteq$ co-NL, hence $NL =$ co-NL (why?).

4. (a) Consider the following algorithm:

Algorithm 3 Decide-PAL(x)

Require: $x \in \{0,1\}^n$.
Ensure: Returns *true* if $x \in PAL$ and $false$ otherwise.

```
for i ← 1 to n do
    if x[i] ≠ x[n + 1 − i] then
        return false
return true
```

Space Analysis:
$O(\log n)$ for $i \Rightarrow O(\log n)$.

Correctness:
($\Leftrightarrow$) $x \in PAL$ if and only if $\forall i\ x[i] = x[n-i+1]$; Decide-PAL returns *true* if and only if $x[i] = x[n-i+1]$ for all i.

(b) Consider the following algorithm:

Algorithm 4 Decide-Not-PAL(x)

Require: $x \in \{0,1\}^n$.
Ensure: Returns $false$ if $x \in PAL$ and *true* otherwise.

```
guess 1 ≤ i ≤ n {actually i ≤ ⌊n/2⌋ is enough}
if x[i] ≠ x[n − i + 1] then
    return true
else
    return false
```

Space Analysis:
$O(\log n)$ for $i \Rightarrow O(\log n)$.

Time Analysis:
One sweep forward to find n, one sweep backward to gather $x[i], x[n-i+1] \Rightarrow O(n)$. As we saw in Exercise 1, incrementing a counter from 0 to n only costs $O(n)$.

Correctness:
($\Leftrightarrow$) $x \notin PAL$ if and only if $\exists i$ such that $x[i] \neq x[n-i+1]$; DECIDE-NOT-PAL returns *true* if and only if for some execution path, i.e., a guess for i, $x[i] \neq x[n-i+1]$.

Note: $PAL \in L \subseteq NL \Rightarrow \overline{PAL} \in \text{co-}NL = NL$ is a simple result of *Immerman-Szelepcsényi theorem*, but we needed the algorithm presented to enforce linear running time.

5. Consider the following algorithm:

Algorithm 5 ADD-BINARY-NUMBERS(x, y)

Require: Two integers $x, y < 2^n$ in binary representation, least-to-most significant bit order.
Ensure: Returns $x + y$ in binary representation, least-to-most significant bit order.

$carry \leftarrow 0$ {can be embodied into the internal state}
for $i \leftarrow 1$ to n **do**
  $output[i] \leftarrow x[i] \oplus y[i] \oplus carry$ { assume 0 if $x[i]$ or $y[i]$ don't exist }
  $carry \leftarrow maj(x[i], y[i], carry)$
$output[n+1] \leftarrow carry$

Space Analysis:
$O(\log n)$ for i, $O(1)$ for $carry \Rightarrow O(\log n)$.

Correctness:
Trivial - we add bit by bit using a full adder.

Note: The algorithm shown uses least-to-most sigfinicant bit order. If we require most-to-least order, this can be achieved as a composition of this algorithm with a rather trivial $O(\log n)$-space REVERSE algorithm, in $O(\log n)$-space (as done with composing log-space reductions).