1. (Exercise 22.1-6 from the text) Let \( A = [a_{ij}] \) be the adjacency matrix of a directed graph \( G = (V, E) \) with no self-loops. Construct an algorithm with asymptotic time complexity \( \Theta(V) \) that finds a vertex \( v \) with properties

(a) out-degree of \( v \) is zero
(b) in-degree of \( v \) is \( |V| - 1 \).

If there is no vertex with these properties, the algorithm should output a statement to that effect; otherwise it outputs the vertex number. Your writeup should include

(a) Pseudocode for the algorithm in the spirit of the examples covered in lecture.
(b) An explanation as to why at most one vertex can have desired properties.
(c) An argument that shows your algorithm is \( \Theta(V) \).

Assume that the adjacency matrix is already in memory when your algorithm starts. Otherwise, there is no way to achieve \( \Theta(V) \) since it could require \( \Theta(V^2) \) operations just to read in the matrix.

2. (Exercise 22.1-7 from the text) The incidence matrix of a directed graph, \( G = (V, E) \), with no self-loops is a \( V \times E \) matrix \( B = [b_{ij}] \) such that

\[
    b_{ij} = \begin{cases} 
    -1, & \text{if edge } j \text{ leaves vertex } i \\
    1, & \text{if edge } j \text{ enters vertex } i \\
    0, & \text{otherwise.} 
    \end{cases}
\]

Let \( B^T \) denote the transpose of \( B \). That is, \( [B^T]_{ij} = b_{ji} \). Let \( C = [c_{ij}] = BB^T \). What is the meaning of each \( c_{ij} \), in terms of edges in the original \( G \).

3. (Variation on Exercise 22.2-8 of the text) Consider a tree rooted at vertex \( r \) as an undirected graph \( G = (V, E) \). For any two vertices \( u, v \in V \), let \( \delta(u, v) \) denote the shortest-path distance from \( u \) to \( v \) (number of links). Define

\[
    D = \max_{u,v \in V} \delta(u, v).
\]

Starting with an adjacency list representation of \( G \) as a directed graph, construct a \( \Theta(V) \) algorithm that computes \( D \). Note that the input adjacency list will contain only edges from a vertex to its children in the tree; it will not contain the reverse edges, although these reverse edges are used in computing the \( \delta(u, v) \) values. Give an argument to show that your algorithm is \( \Theta(V) \). As with the first problem, assume that the adjacency list is already in memory when your algorithm starts. Otherwise, the result may not be \( \Theta(V) \), regardless of the efficiency of your computation, since it requires \( \Theta(V + E) \) operations just to read in the adjacency list, and that may be \( \omega(V) \) if \( E = \omega(V) \).
Solutions.

1. (Exercise 22.1-6 from the text) Let \( A = [a_{ij}] \) be the adjacency matrix of a directed graph \( G = (V,E) \) with no self-loops. Construct an algorithm with asymptotic time complexity \( \Theta(|V|) \) that finds a vertex with in-degree \(|V| - 1\) and out-degree zero, if such a vertex exists. If the vertex does not exist, the algorithm should output a statement to that effect. Give an argument to show that your algorithm is \( \Theta(V) \).

There can be at most one vertex satisfying the condition, because if there were two, one of them would both point to the other and point to no other, a contradiction. Moreover, if vertex \( i \) satisfies the condition, then row \( i \) of the adjacency matrix is all zeros, while column \( i \) is all ones, except for the zero in the diagonal position. We search for such a row-column combination. The following matrix illustrates our strategy which follows a path that remains above the main diagonal and always advances in either a row or column direction, which means that it consults only \( 2|V| \) cells in the adjacency matrix.

```
l_1 l_2 l_3 l_4
0 0 0 0 0 0 0 1 ? ? ? ? ?
```

Starting in the upper left corner of the adjacency matrix, our initial assumption is that vertex 1 is the desired vertex. We scan right in row 1 while we encounter zeros. These zeros confirm our assumption, while the first one disqualifies vertex 1. If we arrive at the right matrix edge before encountering a 1, then vertex 1 satisfies out-degree = 0, and no other vertex can satisfy in-degree = \(|V| - 1\), since vertex 1 does not point to it. It is then a simple matter to check column 1 for the required composition, which requires another \( \Theta(V) \) operations.

If we hit a 1, in column \( k_1 \), before the right matrix edge, we disqualify vertex 1, but also all \( j < k_1 \), because their column has no one in row 1. So, \( k_1 \) is the smallest viable candidate. We proceed downward on column \( k_1 \) so long as we encounters ones. If the first zero occurs on the diagonal, \( k_1 \) remains viable, and we proceed to the right on row \( k_1 \) so long are there are zeros. We are now in the same situation as when we were scanning row 1. If we reach the right edge, \( k_1 \) may be the desired vertex and certainly no other vertex can be so. We then check its row and column completely for compliance, again requiring another \( \Theta(V) \) operations.

If we hit a zero before the diagonal, then \( k_1 \) disqualifies, and the next viable candidate is \( k_1 + 1 \). We drop to the diagonal cell \((k_1 + 1,k_1 + 1)\) and continue scanning zeros to the right, again just as in the first row scan. In this manner we either achieve the right edge with a candidate, or the adjustment to \( k + 1 \) exceeds \( n \), the matrix width, signaling that no vertex possesses the desired properties. We trace a path of horizontal and vertical segments in the upper right triangle of the matrix of length less than or equal to \( 2|V| \). A subsequent check that the surviving candidate has the appropriate row and column properties requires another \( O(|V|) \) operations. Thus the entire algorithm is \( \Theta(|V|) \). In the algorithm below, \( c \) is the candidate vertex, and \( s \) is the scanner. The return is the desired vertex, or a -1 if no such vertex exists. The input is the adjacency matrix \( A \).
function Find(A)
    n = A.width;
    c = 1;
    s = 1;
    dir = 1; // row scan
    while (s <= n) loop
        if (dir == 1)
            if (A[c, s] == 0)
                s = s + 1;
            else
                c = s;
                dir = 2; // column scan
            end if;
        else
            if (A[s, c] == 1)
                s = s + 1;
            else
                dir = 1;
                if (s != c)
                    c = c + 1;
                    s = c;
                end if;
            end if;
        end if;
    end loop;
    if (c > n)
        return -1
    else
        for i in 1 to n loop
            if (A[c, i] > 0)
                return -1;
            end if;
        end loop;
        for i in 1 to n loop
            if ((i != c) && (A[i, c] == 0))
                return -1;
            end if;
        end loop;
        return c;
    end if;
end Find;

2. (Exercise 22.1-7 from the text) The incidence matrix of a directed graph, $G = (V, E)$, with no self-loops is a $|V| \times |E|$ matrix $B = [b_{ij}]$ such that

$$
b_{ij} = \begin{cases} 
-1, & \text{if edge } j \text{ leaves vertex } i \\
1, & \text{if edge } j \text{ enters vertex } i \\
0, & \text{otherwise.}
\end{cases}
$$

Let $B^T$ denote the transpose of $B$. That is, $[B^T]_{ij} = b_{ji}$. Let $C = [c_{ij}] = BB^T$. What is the meaning of each $c_{ij}$, in terms of edges in the original $G$.

Let $|V| = n, |E| = m$. The computation for $c_{ij}$ is

$$
c_{ij} = \sum_{k=1}^{m} b_{ik} [B^T]_{kj} = \sum_{k=1}^{m} b_{ik} b_{jk}.
$$
For each term in the summation, we have

\[ b_{ik}b_{jk} = \begin{cases} 
-1, & \text{edge } k \text{ leaves } i \text{ and enters } j, \text{ or edge } k \text{ leaves } j \text{ and enters } i \\
1, & \text{edge } k \text{ leaves } i \text{ and leaves } j, \text{ or edge } k \text{ enters } i \text{ and enters } j \\
0, & \text{otherwise.}
\end{cases} \]

The middle case is not possible for \( i \neq j \), because each edge has a single source and a single destination. Moreover, since \( G \) has no self-loops, the middle case gives \( b_{ik}^2 = 1 \) for each edge either leaving or entering vertex \( i \).

So, if \( i = j \), each 1-summand corresponds to an edge leaving or entering \( i \). Consequently, \( c_{ii} \) is the number of such edges, the out-degree plus the in-degree. For \( i \neq j \), \(-c_{ij}\) is the number of edges from \( i \) to \( j \) plus the number from \( j \) to \( i \). In summary,

\[ c_{ij} = \begin{cases} 
\text{number of edges incident on } i, & \text{if } i = j \\
-\text{number of edges between } i \text{ and } j, & \text{if } i \neq j.
\end{cases} \]

3. (Variation on Exercise 22.2-8 of the text) Consider a tree rooted at vertex \( r \) as an undirected graph \( G = (V,E) \). For any two vertices \( u,v \in V \), let \( \delta(u,v) \) denote the shortest-path distance from \( u \) to \( v \) (number of links). Define

\[ D = \max_{u,v \in V} \delta(u,v). \]

Starting with an adjacency list representation of \( G \) as a directed graph, construct a \( \Theta(|V|) \) algorithm that computes \( D \). Note that the input adjacency list will contain only edges from a vertex to its children in the tree; it will not contain the reverse edges, although these reverse edges are used in computing the \( \delta(u,v) \) values. Give an argument to show that your algorithm is \( \Theta(V) \).

In general, the diameter of a subtree rooted at \( v \) comes from a path inside one of its children or a path originating in one of its children, continuing on to \( v \) and then descending through a second child. In the last case, the diameter is \( 2 + \text{the heights of the two tallest children} \). If there is just one child, it would be \( 1 + \text{the height of the child} \). The code below implements this observation. We see that a leaf vertex costs 2 instructions, while each interior vertex costs \( 6 + 4 \times \text{its child count} \). The 6 includes the recursive call instruction but not the work done within the recursive execution.

\[ \text{TreeDiameter(Tree t) returns Record(int diameter, int height) \{} \]
\[ \text{if (Adjacency list of t is empty)} \]
\[ \text{return (0, 0);} \]
\[ \text{H = 0; h = 0; // largest two subtree heights} \]
\[ d = 0; \]
\[ \text{for v in adjacency list of t loop} \]
\[ (dc, hc) = \text{TreeDiameter(v);} \]
\[ (H, h) = \text{largest two of (H, h, hc);} \]
\[ d = \text{max(d, dc, H + h + (h > 0 ? 2 : 1)); // h = 0 implies one child} \]
\[ \text{return (d, H + 1);} \]

Let \( V_0 \) be the number of leaves, \( V_1 \) be the number of interior vertices. Since each child of an interior vertex is associated with a distinct edge in \( E \), the total cost is

\[ T = 2V_0 + 6V_1 + 4E \leq 6V_0 + 6V_1 + 6E = 6V + 6E = 6(V + E) = O(V + E) \]
\[ T = 2V_0 + 6V_1 + 4E \geq 2V_0 + 2V_1 + 2E = 2(V + E) = \Omega(V + E). \]

We conclude \( T = \Theta(V + E) \).