Problem 1:

Idea: Let \( n = |V| \) and \( m = |A| \). There are \( n \) subproblems, with each subproblem consisting of the subgraph formed on the first \( i \) vertices \((v_1, v_2, \ldots, v_i)\), for \( i = 1, 2, \ldots, n \). An array \( M[1..n] \) is used to store the solution to each of the subproblems. Here, \( M[i] \) stores the weight of the maximum weight directed path ending at \( v_i \). Once we have the values for all the \( n \) entries in \( M \), the solution to the problem is the largest value in \( M \).

The entries of \( M \) can be computed as follows. Since \( v_1 \) cannot have any incoming edges, we have \( M[1] = 0 \). Now assuming that we have computed the first \( i \) entries of \( M \), \( M[i+1] \) is given by

\[
M[i + 1] = \max\{M[j] + w(v_j, v_{i+1}) : (v_j, v_{i+1}) \text{ is a directed edge in } A\}
\]

The above equation is a simple consequence of the fact that for each directed path that ends at \( v_{i+1} \), the last edge in the path is an incoming edge to \( v_{i+1} \).

High-Level Description of the Algorithm:

1. \( M[1] = 0 \).
2. for \( i = 1 \) to \( n - 1 \) do
   (a) CurrentMax = 0.
   (b) for each directed edge \((v_j, v_{i+1}) \in A\) do
       (i) Temp = \( M[j] + w(v_j, v_{i+1}) \).
       (ii) if (CurrentMax < Temp) then CurrentMax = Temp.
   (c) \( M[i + 1] = \) CurrentMax.
3. Output the largest of the \( n \) values in \( M \).

Running Time Analysis: Step 1 uses \( O(1) \) time while Step 3 uses \( O(n) \) time. To analyze the time for Step 2, let \( \Delta(v_i) \) denote the indegree of vertex \( v_i \) (i.e., the number of incoming edges to \( v_i \)). In Step 2, we consider all the incoming edges to vertex \( v_{i+1} \) and spend \( O(1) \) time for each such edge. Thus, for vertex \( v_{i+1} \), the time spent in Step 2 is \( O(\Delta(v_{i+1})) \). Therefore, the total time spent in Step 2 for all the vertices is \( O(\sum_{i=1}^{n-1} \Delta(v_{i+1})) \). It is easy to see that \( \sum_{i=1}^{n-1} \Delta(v_{i+1}) = |A| = m \). Therefore, Step 2 runs in \( O(m) \) time and the whole algorithm has a running time of \( O(n + m) \).
Problem 2: Consider the following example.

\begin{center}
\begin{tikzpicture}
\node (v1) at (0,0) {$v_1$};
\node (v2) at (1,0) {$v_2$};
\node (v3) at (2,0) {$v_3$};
\node (v4) at (3,0) {$v_4$};
\node (v5) at (4,0) {$v_5$};
\draw (v1) -- (v2);
\draw (v2) -- (v3);
\draw (v3) -- (v4);
\draw (v4) -- (v5);
\end{tikzpicture}
\end{center}

The set \( \{v_2, v_4\} \) is an optimal dominating set of size 2 for the above graph. (It is easy to verify that no single vertex by itself is a dominating set.)

When we run greedy, vertices \( v_2, v_3 \) and \( v_4 \) all have degree = 2. Because ties are broken arbitrarily, we can force the greedy algorithm to pick \( v_3 \) first. When this is done, vertices \( v_2 \) and \( v_4 \) are both eliminated and the algorithm is forced to pick both the remaining vertices, namely \( v_1 \) and \( v_5 \). Therefore, the dominating set \( \{v_1, v_3, v_5\} \) picked by the greedy algorithm has 3 vertices, which is not optimal.

Problem 3:

Idea: We successively remove each vertex whose degree is less than \( k \) from the graph. If we reach a stage where we have a subgraph in which every vertex has degree \( \geq k \), we stop and output the set of vertices of that subgraph. The reason why this is an optimal solution is that none of the vertices that were removed can be part of an optimal solution.

High-Level Description of the Algorithm:

1. Compute the degree of each vertex in the graph.
2. while there is a vertex \( v \in V \) with degree \( < k \) do
   (a) For each vertex \( w \) such that \( \{v, w\} \) is an edge in the remaining graph, decrement the degree of \( w \) by 1.
   (b) Remove \( v \) from \( V \) and the edges incident on \( v \) from \( E \).
3. Output the set \( V \) of remaining vertices as the optimal solution.

Proof of correctness: Suppose the algorithm removes \( r \) vertices and that the sequence in which vertices get removed is \( \langle v_1, v_2, \ldots, v_r \rangle \). We prove by induction on \( r \) that none of the vertices removed by the algorithm can be part of a feasible solution (i.e., subgraph in which each vertex has degree \( \geq k \)).

Basis: Consider the first vertex \( v_1 \) removed by the algorithm. The vertex was removed because its degree is less than \( k \). Thus, \( v_1 \) can’t be part of any feasible solution.
**Inductive hypothesis:** Assume that for some \( i \geq 1 \), the set of vertices \( S_i = \{v_1, \ldots, v_i\} \) can’t part of any feasible solution.

**To prove:** Node \( v_{i+1} \) can’t be part of any feasible solution.

**Proof:** Node \( v_{i+1} \) was removed because its degree at that stage was less than \( k \). If \( v_{i+1} \) is not adjacent to any of the vertices in \( S_i \), then the degree of \( v_{i+1} \) is less than \( k \) in the original graph itself and so it can’t be part of a feasible solution. If \( v_{i+1} \) is adjacent to one or more vertices in \( S_i \), then the degree of \( v_{i+1} \) became less than \( k \) because of the removal of some vertices from \( S_i \). Since none of the vertices in \( S_i \) is part of a feasible solution, it follows that \( v_{i+1} \) can’t be part of a feasible solution either. This completes the proof.

**Running time:** Let \( n = |V| \). We assume that the graph is represented by an adjacency matrix. (For a graph with \( n \) vertices, this is an \( n \times n \) Boolean matrix \( M \), where \( M[i, j] = 1 \) if \( \{v_i, v_j\} \) is an edge and 0 otherwise.)

The degree of each vertex can be computed in \( O(n) \) time (we just need to count the number of 1’s in the corresponding row). So, Step 1 runs in \( O(n^2) \) time. By keeping a linked list of vertices with degree less than \( k \), we can find such a vertex in \( O(1) \) time. Deleting a vertex \( v_i \) and the edges incident on \( v_i \) can be done in \( O(n) \) time (since we only need to change each 1 entry in row \( i \) to 0). When we do this, we can also decrement (by 1) the degree of each vertex \( v_j \) that is adjacent to \( v_i \); if the degree of \( v_j \) becomes less than \( k \), then it can be added to the linked list in \( O(1) \) time. Thus, the time needed for each iteration of Step 2 is \( O(n) \). Since the loop runs at most \( n \) times, the total time for Step 2 is \( O(n^2) \). Step 3 runs in \( O(n) \) time. Therefore, the overall running time is \( O(n^2) \).

**Note:** If we use the adjacency list representation of the graph, the algorithm can be be implemented to run in \( O(|V| + |E|) \) time.

---

**Problem 4:**

**Idea:** Sort the tasks in increasing order of deadlines and schedule them in that order. We will show that if this algorithm doesn’t produce a feasible schedule (i.e., one in which every task is completed by its deadline), then there is no feasible schedule.

**High-Level Description of the Algorithm:**

1. Sort the tasks in increasing order of deadlines. Without loss of generality, let \( \langle T_1, T_2, \ldots, T_n \rangle \) denote this sorted order.
2. If the above schedule is feasible, then output “Yes”; otherwise, output “No”.
Proof of correctness: We will prove using an exchange argument that if there is a feasible schedule, then the schedule produced by the algorithm is also feasible.

Let $S = \langle T_1, T_2, \ldots, T_n \rangle$ denote the schedule produced by the algorithm. Let $S' = \langle T_{i_1}, T_{i_2}, \ldots, T_{i_n} \rangle$ denote a feasible schedule. Suppose for some $k \geq 0$, the first $k$ tasks in $S$ are the same as those in $S'$. (Since we allow $k$ to be 0, such a $k$ exists.) We will show that we can create another feasible schedule $S''$ by exchanging a pair of tasks in $S'$; further, by this exchange, the first $k + 1$ tasks in $S''$ will be the same as those in $S$. By repeating this process, we can successively modify $S'$ so that it completely agrees with $S$.

In $S$ and $S'$, tasks in position $k + 1$ are respectively $T_{k+1}$ and $T_{i_{k+1}}$. Since $S$ and $S'$ agree in the first $k$ tasks, task $T_{k+1}$ occurs after task $T_{i_{k+1}}$ in $S'$. Let $i_j$ denote the index of task $T_{k+1}$ in $S'$; that is, task $T_{k+1}$ occurs as task $T_{i_j}$ in $S'$. Construct $S''$ from $S'$ by exchanging $T_{i_{k+1}}$ and $T_{i_j}$. We now prove that $S''$ is also feasible given that $S'$ is feasible.

First note that the set of tasks in positions $k + 1$ through $n$ in $S$ and $S'$ are equal. (However, the tasks are in different orders in the two schedules.) Because of sorting, task $T_{k+1}$ has the smallest deadline ($d_{k+1}$) among all these tasks. Let $f$ denote the finishing time of $T_{i_j} = T_{k+1}$ in $S'$. Since $S'$ is feasible, $f \leq d_{k+1}$. Thus, for every task in positions $k + 1$ through $j - 1$ in $S'$, the deadline is at least $f$. Therefore, when we exchange $T_{i_{k+1}}$ and $T_{k+1}$, none of the deadlines is violated. In other words, $S''$ is also feasible. This completes the proof.

Running time: Sorting the tasks can be $O(n \log n)$ time. Thus, Step 1 runs in $O(n \log n)$ time. Once we have the sorted order, the completion time of all tasks can be computed in $O(n)$ time. Therefore, the feasibility of the schedule can be determined in $O(n)$ time. In other words, Step 2 runs in $O(n)$ time. Therefore, the overall running time is $O(n \log n)$. 