Each problem is worth 10 points:

1. Given two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$, we want to calculate the length of the longest common substring, namely the largest $k$ for which there are $i, j$ such that $x_i x_{i+1} \ldots x_{i+k-1} = y_j y_{j+1} \ldots y_{j+k-1}$. Show how to do this in time $O(mn)$.

**Solution.** Denote $opt(i, j)$ as the longest common substrings that starts at $x_i$ and $y_j$. Then we have the following rule:

$$
opt(i, j) = \begin{cases} 
1 + opt(i + 1, j + 1) & \text{if } x_i = y_j \\
0 & \text{else.}
\end{cases}
$$

And the boundary conditions are

$$
opt(i, n) = \begin{cases} 
1 & \text{if } x_i = y_n \\
0 & \text{else.}
\end{cases}
$$

$$
opt(m, j) = \begin{cases} 
1 & \text{if } x_m = y_j \\
0 & \text{else.}
\end{cases}
$$

So we have the following algorithm:

```plaintext
Input: strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$
Result: length of longest common strings
Initialize table OPT[m][n]
for $i=m,m-1,\ldots,1$ do
    for $j=n,n-1,\ldots,1$ do
        if $i == m$ or $j == n$ then
            $OPT[i][j] = (x_i == y_j)$
        end
        else if $x_i == y_j$ then
            $OPT[i][j] = 1 + OPT[i+1][j+1]$  
        end
        else
            $OPT[i][j] = 0$
        end
    end
end
return max($OPT[i][j]$)
```
2. You are a large corporation that wants to open a chain of stores along a highway. There are \( n \) possible locations, which are at mileposts \( m_1, \ldots, m_n \) on the highway. At each location \( m_i \), you may open one store, which will give you an expected profit of \( p_i \). However, if you open stores at \( m_i, m_j \), then these stores must be at least \( k \) miles apart (i.e. \( |m_i - m_j| \geq k \)). Give an efficient algorithm to find the optimal locations to open stores on input \( k, p_1, \ldots, p_n \) and \( m_1, \ldots, m_n \). (For full credit it is enough to calculate the maximum expected profit from the best solution).

**Solution.** First, observe that if every \( p_i < 0 \), then we open zero stores amounting to an expected profit of 0. Second, we can drop all locations \( i \) for which the expected profit \( p_i < 0 \). This is because dropping these locations will not change the value of the optimum expected profit; if the optimum set of locations contains one with negative expected profit, then we can remove that location to increase the overall profit. In what follows, the expected profits satisfy \( p_1 \geq 0, \ldots, p_n \geq 0 \).

We first sort according to the mileposts and they satisfy \( m_1 \leq m_2 \leq \ldots \leq m_n \). For every \( m_i \), let \( i^* \) be the index of the smallest \( m_j \) that is at least \( m_i + k \). In other words, \( i^* \) is the smallest element in \( \{ j \mid m_j \geq m_i + k \} \). If no such element exists, then we take \( i^* = n + 1 \). The pseudocode is given below.

```plaintext
Input: \( m_1, \ldots, m_n \) and \( k, p_1, \ldots, p_n \)
Result: Optimum expected profit

Set \( A[n+1] = 0 \) and \( A[n] = p_n \).
for \( i = n-1, n-2, \ldots, 1 \) do
    \[ A[i] = \max\{p_i + A[i^*], A[i+1]\}. \]
end
return \( A[1] \).
```

**Runtime:** First, it takes \( O(n \log n) \) time to sort the \( m_i \)'s and then compute \( i^* \) for each \( i = 1, 2, \ldots, n \). Second, it takes constant time to set each entry in \( A \), which amount to a total of \( O(n) \) time. Putting together these two observations, we can conclude that the running time is \( O(n \log n) \).

**Proof of Correctness.** We will prove it by induction on \( n \). Let \( OPT \) denote the optimal set of locations to open the stores. If \( n = 1 \), then the optimum is just choosing the only
location which the algorithm finds in the step where it assigns \( A[n] = p_n \). We now prove it for arbitrary \( n \) by considering two cases.

(a) \( m_1 \in OPT \): By the induction hypothesis, the algorithm will find the maximum expected profit in \( m_1, \ldots, m_n \). Since, \( p_1 + A[1^*] \) is considered, the algorithm will output the maximum expected profit correctly.

(b) \( m_1 \notin OPT \): By the induction hypothesis, the algorithm will find the maximum expected profit in \( m_2, \ldots, m_n \). Since \( A[2] \) is a candidate for the maximum, the algorithm will output the maximum expected profit correctly.

3. Suppose we are given a flow network, where instead of capacities on edges, each internal vertex has a capacity on the total flow that is allowed to pass through it. So for each vertex \( v \), there is a non-negative integer \( c_v \), and the flow must satisfy \( f^m(v) \leq c_v \). Each edge can carry an arbitrary amount of flow. Give a polynomial time algorithm to find the maximum flow in such a network. (Hint: try to convert the problem into a flow network of the type we are used to.)

**Solution:** The polynomial time algorithm will generate a new standard flow network \( G' \) as follows.

(a) \( G' \) has a start vertex \( s \) and a sink vertex \( t \).
(b) For every vertex \( u \) of \( G \) with capacity \( c \), \( G' \) has two vertices \( u_0, u_1 \), and an edge from \( u_0 \) to \( u_1 \) of capacity \( c \).
(c) For every edge \((s, u)\) in \( G \), we add an edge \((s, u_0)\) to \( G \) with infinite capacity.
(d) For every edge \((u, v)\) in \( G \) between intermediate vertices, we add an edge \((u_1, v_0)\) to \( G' \) of infinite capacity.
(e) For every edge \((v, t)\) in \( G \), add an edge \((v_1, t)\) to \( G' \) of infinite capacity.

The algorithm then finds the max-flow in \( G' \) using the polynomial time algorithm discussed in class.

To see that the algorithm is correct, we prove that there is a valid flow of value \( v \) in \( G \) if and only if there is a flow of value \( v \) in \( G' \). Suppose there is a flow of value \( v \) in \( G \). Obtain a flow in \( G' \) by setting the flow value for every edge \((u_1, v_0)\) to be the same as the flow on \((u, v)\) in \( G \), and the flow value for every edge \((s, u_0)\) to be the same as the flow on \((s, u)\) in \( G \), and the flow value for every edge \((v_1, t)\) to be the same as the value on \((v, t)\). Finally set the flow value on every edge \((u_0, u_1)\) to be the same as the total flow into the vertex \( u \). This flow in \( G' \) respects the capacities and has the same value as the flow in \( G \).

Suppose there is a flow of value \( v \) in \( G' \). Then we obtain a flow in \( G \) by setting the flow value of \((s, u)\) to be the same as the flow of \((s, u_0)\), the flow of \((u, v)\) to be the same as the flow on \((u_1, v_0)\) and the flow on \((v, t)\) to be the same as the flow on \((v_1, t)\). Once again, this gives a valid flow that respects the capacity constraints in \( G' \). Thus, the value of the maximum flow in \( G \) is the same as the maximum flow in \( G' \), and the algorithm is correct.

4. Draw out a maximum \( s - t \) flow for the graph below, and the corresponding residual graph \( G_f \). What is the minimum cut that corresponds to this max flow?
Solution. Original Graph:

Residual Graph:
Max $s - t$ flow:
Min-Cut: \{s, a, b, c\}, \{d, t\}. The capacity of the Min-Cut is 12, same as the max-flow.

**Common Errors.** (a) Not showing all parts of the flow graph.
(b) Tracing the min-cut incorrectly in the final residual graph.