**Question 1** (Maximum Independent Set, 30 points). Compute the size of the maximum independent set of the graph below:

![Graph Image]

[Note: you do not need to show your work, but doing so will be useful for assigning partial credit if you make a mistake.]

The answer is 10. We compute this by recursively computing the size of the maximal independent set of the subtree defined by each node. This is the maximum of the sum of the values at the children or one more than the sum of the values at the grandchildren. The computed values are below:

![Computed Values Image]
Question 2 (Road Trip III, 35 points). Once again Thomas is travelling along a highway with stops 1, 2, . . . , n in that order trying to get from stop 1 to stop n. This time Thomas has realized that the number of times he needs to stop for gas is less important than the total amount of money that he needs to spend on it. The \(i^{th}\) station has a value \(x_i\), the number of miles down the road that it is (with \(x_1 < x_2 < \ldots < x_n\)), and a price \(p_i\) which is the amount charged for enough gas for Thomas to drive a mile. Assume that Thomas starts with no gas at stop 1, but can store an unlimited amount in his tank. Give a polynomial time algorithm to compute the minimum amount that Thomas would need to spend on gas to get to stop n.

At each stop whose price is cheaper than any previous price, Thomas should pick up exactly enough gas to get to either the end or to the next stop with even cheaper gas (whichever comes first). To prove that this works, imagine that Thomas could take an unlimited amount of gas from any station, putting it into a can labelled with the station name and only had to pay for it when he used it. Clearly Thomas would want to only use gas from the cheapest station he had yet encountered. This is achieved by this solution. An efficient implementation is as follows:

\[
\begin{align*}
\text{Total} &= 0 \\
\text{MinCost} &= p_1 \\
\text{for } i &= 1 \text{ to } n-1 \\
& \quad \text{Total} = \text{Total} + \text{MinCost}*(x_{i+1}-x_i) \\
& \quad \text{if } p_{i+1} < p_i \\
& \quad \quad \text{MinCost} = p_{i+1} \\
\text{return Total}
\end{align*}
\]

The total runtime of this algorithm is clearly \(O(n)\).
Question 3 (String Smoothing, 35 points). Given a string \( z_1 z_2 \ldots z_n \) the number of breaks is the number of indices \( i \) so that \( z_i \neq z_{i+1} \). Given a string \( X = x_1 x_2 \ldots x_n \), Ivy would like to alter at most \( k \) of the characters to obtain a new string \( Y = y_1 y_2 \ldots y_n \) so that the new string has as few breaks as possible. Give a polynomial time algorithm to compute given \( X \) and \( k \), the fewest number of breaks in \( Y \) that Ivy can achieve.

We use a dynamic program. For each character \( C \), index \( i \), and integer \( m \) we let \( FB(C, i, m) \) be the fewest possible breaks attainable by a string that starts with \( C \), and is followed by some string \( s_i s_{i+1} \ldots s_n \) where \( s_i s_{i+1} \ldots s_n \) is obtained by changing at most \( m \) of the characters in \( x_i x_{i+1} \ldots x_n \). We claim that:

If \( x_i = C \), then \( FB(C, i, m) = FB(C, i + 1, m) \). This is because in any valid solution, replacing \( s_i \) with \( x_i = C \) only decreases the number of changes and breaks. Therefore, we can assume that \( s_i = C \), and the best we can do is the fewest number of breaks in a string of the form \( C s_{i+1} \ldots s_n \).

If \( x_i \neq C \), then \( FB(C, i, m) \) is the minimum of \( FB(x_i, i + 1, m) + 1 \) and \( FB(C, i + 1, m - 1) \). This is because in any sequence either \( x_i \) isn’t changed, in which case there is a break \( C \) to \( x_i \) plus the number of breaks in a string of the form \( x_i s_{i+1} \ldots s_n \) with at most \( m \) edits (for which the best possible number of breaks is \( FB(x_i, i + 1, m) + 1 \), or \( x_i \) is changed. In this case, changing \( x_i \) to \( C \) gives as few breaks as any other choice, and so we have as many breaks as a string of the form \( C s_{i+1} \ldots s_n \) with at most \( m - 1 \) edits (for which the best possible number of breaks is \( FB(C, i + 1, m - 1) \)). The best we can do overall is the minimum of these two numbers.

Note that these recursions work when \( i = n \) if \( FB(C, n + 1, m) \) is defined to be 0, since there are no extra breaks after possibly the one between \( C \) and \( s_n \).

The algorithm is now as follows:

1. If \( k \geq n \) return 0.
2. Let \( A \) be an alphabet consisting of all characters appearing in \( X \), plus an additional null character \( \emptyset \).
3. Let \( F \) be an array with indices \( F(C, i, m) \) with \( C \in A \) and \( 1 \leq i \leq n + 1 \) and \( 0 \leq m \leq k \).
4. For each \( C, m \), let \( F[C, n + 1, m] = 0 \).
5. For \( i \) decreasing from \( n \) to 1
   - For each \( C, m \) let
     \[
     F[C, i, m] = \begin{cases} 
     F[C, i + 1, m] & \text{if } C = x_i \\
     \min\{F[x_i, i + 1, m] + 1, F[C, i + 1, m - 1]\} & \text{else}
     \end{cases}
     \]
6. Return \( F[\emptyset, 1, k] - 1 \)

The runtime of this algorithm is dominated by the for loop. It has \( O(|A|nk) \) iterations each taking constant time. Therefore, the final runtime is \( O(n^2k) \).

To show correctness, we note by induction that each entry filled in to \( F[C, i, m] \) is the correct value of \( FB(C, i, m) \) (where \( FB(C, n + 1, m) \) is defined to be 0). To cover the base case, we note that all the entries with \( i = n + 1 \) are assigned correctly. For later values, the entry assigned is the value given by the recurrence above (since by the inductive hypothesis the previous values of \( F \) are correct). Therefore, the final entry in \( F[\emptyset, i, k] \) is the best possible number of breakpoints in a valid string \( \emptyset Y \), which is one more than the best number of breakpoints for just \( Y \).