1. Describe and analyze an algorithm that computes the length of the shortest summary of two given strings $A[1..m]$ and $B[1..n]$. The delimiter length $\Delta$ is also part of the input to your algorithm.

**Solution (first delimiter):** Fix the input strings $A$ and $B$ and the delimiter penalty $\Delta$. To simplify bookkeeping, add unique sentinel characters $A[m + 1] \neq B[n + 1]$ to the ends of the strings.

For all integers $i$ and $j$ and any delimiter $prev$, let $Short(i, j, prev)$ denote the minimum length of any summary of the suffixes $A[i..m]$ and $B[j..n]$ that begins with $prev$, excluding the length of this initial delimiter. Intuitively, as we recursively build the optimal summary from left to right, we remember the last delimiter that we used so far.

This function satisfies the following recurrence:

$$\begin{align*}
Short(i, j, prev) = \begin{cases} 
\infty & \text{if } i > m + 1 \text{ or } j > n + 1 \\
0 & \text{if } i = m + 1 \text{ and } j = n + 1 \\
\min \left\{ Short(i + 1, j, \text{▲}) + 1, \right. \\
\left. Short(i + 1, j, \text{▼}) + 1 + \Delta, \right. \\
\left. Short(i + 1, j, \text{●}) + 1 + \Delta \right\} & \text{if } prev = \text{▲} \\
\min \left\{ Short(i, j + 1, \text{▲}) + 1 + \Delta, \right. \\
\left. Short(i, j + 1, \text{▼}) + 1, \right. \\
\left. Short(i, j + 1, \text{●}) + 1 + \Delta \right\} & \text{if } prev = \text{▼} \\
\min \left\{ Short(i + 1, j + 1, \text{▲}) + 1 + \Delta, \right. \\
\left. Short(i + 1, j + 1, \text{▼}) + 1 + \Delta, \right. \\
\left. Short(i + 1, j + 1, \text{●}) + 1 \right\} & \text{if } prev = \text{●} \text{ and } A[i] = B[j] \\
\infty & \text{if } prev = \text{●} \text{ and } A[i] \neq B[j]
\end{cases}
\end{align*}$$

The initial base case prevents us from writing the sentinel characters into the recursively constructed summary; without this case, we would need to test in each recursive case that one suffix or both suffixes are non-empty. The second base case correctly handles the case where both suffixes are empty. In each recursive case, we have three choices: continue in the current “mode” or switch to a different delimiter (at a cost of $\Delta$). The last case ensures that we only consider summaries starting with $\text{●}$ if the two suffixes start with the same symbol (since we don’t check for that condition before calling $Short(\cdot, \cdot, \text{●})$).

If $A$ and $B$ are empty, the optimal (in fact only) summary has length zero. Otherwise, the shortest summary of $A$ and $B$ has length

$$\Delta + \min \{ Short(1, 1, \text{▲}), Short(1, 1, \text{▼}), Short(1, 1, \text{●}) \}.$$  

There are three possibilities for the initial delimiter, each of which costs $\Delta$; once we’ve paid for that delimiter, the remaining cost is computed by the corresponding call to $Short(1, 1, \cdot)$.

We can memoize the recursive function $Short$ into an $m \times n \times 3$ array, indexed by $i$, $j$, and the delimiters. We can fill this array by decreasing $i$ in the outermost loop, decreasing $j$ in the middle loop, and the delimiters in arbitrary order in the innermost loop. The resulting algorithm runs in $O(mn)$ time.
Solution (first delimiter, take two): Fix the input strings $A$ and $B$ and the delimiter penalty $\Delta$. For all integers $i$ and $j$, we define four different functions that denote the length of the shortest summary of the suffixes $A[i..m]$ and $B[j..n]$ satisfying certain constraints.

- $\text{Short}(i, j)$ — shortest summary (no constraints)
- $\text{Short}^\triangle(i, j)$ — shortest summary beginning with $\triangle$
- $\text{Short}^\nabla(i, j)$ — shortest summary beginning with $\nabla$
- $\text{Short}^{\bullet}(i, j)$ — shortest summary beginning with $\bullet$

These functions satisfy the following mutual recurrences:

$$\text{Short}(i, j) = \begin{cases} 0 & \text{if } i > m \text{ and } j > n \\ \min \left\{ \begin{array}{l} \text{Short}^\triangle(i, j) \\ \text{Short}^\nabla(i, j) \\ \text{Short}^{\bullet}(i, j) \end{array} \right\} & \text{otherwise} \end{cases}$$

$$\text{Short}^\triangle(i, j) = \begin{cases} \infty & \text{if } i > m \\ 1 + \min \left\{ \begin{array}{l} \text{Short}^\triangle(i + 1, j) \\ \Delta + \text{Short}(i + 1, j) \end{array} \right\} & \text{otherwise} \end{cases}$$

$$\text{Short}^\nabla(i, j) = \begin{cases} \infty & \text{if } j > n \\ 1 + \min \left\{ \begin{array}{l} \text{Short}^\nabla(i, j + 1) \\ \Delta + \text{Short}(i, j + 1) \end{array} \right\} & \text{otherwise} \end{cases}$$

$$\text{Short}^{\bullet}(i, j) = \begin{cases} \infty & \text{if } i > m \text{ or } j > n \text{ or } A[i] \neq B[j] \\ 1 + \min \left\{ \begin{array}{l} \text{Short}^{\bullet}(i + 1, j + 1) \\ \Delta + \text{Short}(i + 1, j + 1) \end{array} \right\} & \text{otherwise} \end{cases}$$

In the first recurrence, unless both suffixes are empty, we choose the starting delimiter but do not pay for it yet. In each of the other recurrences, if the suffixes do not permit a summary that starts with the corresponding delimiter, we return $\infty$ as an error code. Otherwise, we write the next symbol into the current block of the summary (at a cost of 1) and then decide whether to end the current block (at an additional cost of $\Delta$). Thus, we pay for each initial delimiter in the resulting summary at the end of its block.

We can memoize each of these functions into a two-dimensional array. We can fill all four arrays with a pair of nested loops, decreasing $i$ in the outer loop and decreasing $j$ in the inner loop. Inside the loops, we evaluate $\text{Short}^\triangle(i, j)$, $\text{Short}^\nabla(i, j)$, and $\text{Short}^{\bullet}(i, j)$ before evaluating $\text{Short}(i, j)$. The resulting algorithm runs in $O(mn)$ time. \hfill \blacksquare
Solution (first chunk, slower): Fix the input strings $A$ and $B$ and the delimiter cost $\Delta$. For all integers $i$ and $j$, let $\text{Short}(i, j)$ denote the minimum length of any summary of the suffixes $A[i..m]$ and $B[j..n]$. This function can be computed by the following recursive algorithm:

\[
\text{Short}(i, j) :=
\begin{cases}
0 & \text{if } i > m \text{ and } j > n \\
\infty & \text{otherwise}
\end{cases}
\]

for $\ell \leftarrow 1$ to $m - i$
\[
\text{short} \leftarrow \min \{ \text{short}, \, \Delta + \ell + \text{Short}(i + \ell + 1, j) \}
\]

for $\ell \leftarrow 1$ to $n - j$
\[
\text{short} \leftarrow \min \{ \text{short}, \, \Delta + \ell + \text{Short}(i, j + \ell + 1) \}
\]

$\ell \leftarrow 1$

while $i + \ell \leq m$ and $j + \ell \leq n$ and $A[i + \ell] = B[j + \ell]$
\[
\text{short} \leftarrow \min \{ \text{short}, \, \Delta + \ell + \text{Short}(i + \ell + 1, j + \ell + 1) \}
\]

$\ell \leftarrow \ell + 1$

return short

Unless both suffixes are empty, there are three possibilities for the first delimiter in the optimal summary. For each of these choices, the algorithm considers all possible substrings that can follow that initial delimiter. The first for-loop considers chunks starting with ▲; the second for-loop considers chunks starting with ▼; and the last while loop considers chunks starting with †.

We can memoize the function Short into an $m \times n$ array, with rows indexed by $i$ and columns indexed by $j$. Each entry $\text{Short}[i, j]$ depends only on other entries with larger indices. Thus, we can fill this array with two nested loops, by decreasing $i$ in the outer loop, and decreasing $j$ in the inner loop. Filling each entry requires $O(m + n)$ time, so the resulting algorithm runs in $O(mn(m + n))$ time.

Rubric: The third solution is worth at most 3 points out of 4.

All these solutions are more detailed than necessary for full credit. These are not the only correct solutions. In particular, there are similar solutions whose cases depend on the last delimiter in the optimal summary.

Full credit requires that the English description of the recursive function precisely matches what the function actually computes. Any full-credit solution must correctly handle the following boundary/base cases:

- Both input strings are empty.
- Exactly one of the input strings is empty.
- In some recursive call, both suffixes (or prefixes, or whatever) are empty.
- In some recursive call, exactly one suffix (or prefix, or whatever) is empty.

No penalty for silently assuming (1) that $\Delta$ cannot be negative and (2) that at least one regular character must follow each delimiter. If $\Delta$ is negative and consecutive delimiters are forbidden, the optimal solution has cost $(m + n)(\Delta + 1)$; for example: ▲K▲I▲T▲E▲N▲V▲K▲V▲N▲I▲V▲T▲V▲I▲V▲N▲V▲G. If $\Delta$ is negative and consecutive delimiters are allowed, then there is no optimal solution, because adding another † to either end of a summary makes it shorter. Finally, if $\Delta = 0$, there is always an optimal summary with no consecutive delimiters, and if $\Delta > 0$, every optimal summary has no consecutive delimiters.
2. Suppose you are given a sequence of positive integers separated by plus (+) and minus (−) signs. Describe and analyze an algorithm to compute the maximum possible value the expression can take by adding parentheses.

Solution: Suppose the input is an array \( S[0..2n] \), where \( S[i] \) is a positive integer when \( i \) is even, and \( S[i] \) is either + or - when \( i \) is odd. For all even indices \( i \leq j \), we define two functions.

- \( \text{MaxV}(i, k) \) is the largest value that can be obtained from the substring \( S[i..k] \) by adding parentheses.
- \( \text{MinV}(i, k) \) is the smallest value that can be obtained from the substring \( S[i..k] \) by adding parentheses.

These functions satisfy the following mutual recurrences:

\[
\text{MaxV}(i, k) = \begin{cases} 
    S[i] & \text{if } i = k \\
    \max_{i < j < k \text{ and } \text{j odd}} \left( \begin{array}{l}
        \text{MaxV}(i, j - 1) + \text{MaxV}(j + 1, k) \quad \text{if } S[j] = + \\
        \text{MaxV}(i, j - 1) - \text{MinV}(j + 1, k) \quad \text{if } S[j] = -
    \end{array} \right) & \text{otherwise}
\end{cases}
\]

\[
\text{MinV}(i, k) = \begin{cases} 
    S[i] & \text{if } i = k \\
    \min_{i < j < k \text{ and } \text{j odd}} \left( \begin{array}{l}
        \text{MinV}(i, j - 1) + \text{MinV}(j + 1, k) \quad \text{if } S[j] = + \\
        \text{MinV}(i, j - 1) - \text{MaxV}(j + 1, k) \quad \text{if } S[j] = -
    \end{array} \right) & \text{otherwise}
\end{cases}
\]

We can memoize these functions into a single two-dimensional array \( \text{Values}[0..2n, 0..2n] \) of records with two fields \( \text{max} \) and \( \text{min} \), where \( \text{Values}[i, k].\text{max} = \text{MaxV}(i, k) \) and \( \text{Values}[i, k].\text{min} = \text{MinV}(i, k) \). Each entry in this array depends on all other entries directly below or directly to the left. Thus, we can fill the array in reverse row-major order: decreasing \( i \) in the outer loop, and decreasing \( j \) in the inner loop.

Since each entry \( \text{Values}[i, k] \) requires \( O(n) \) time to compute from earlier entries, the resulting algorithm runs in \( O(n^3) \) time.

Rubric: A full-credit solution must specify the input format; whether to use one array (as here) or two (one with numbers and the other with signs) is a matter of taste. Similarly, is a matter of taste whether to memoize into a single two-dimensional array with two fields per entry, or two two-dimensional arrays, or a three-dimensional array where one dimension has size 2, as long as the solution is clear.
3. Suppose you are given a rooted tree $T$. You want to label each node in $T$ with an integer 1, 2, or 3, such that every node has a different label from its parent. The cost of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree $T$.

Solution: Let $Cost(v, \ell)$ denote the minimum cost of any labeling of the subtree rooted at $v$ such that $\text{label}(v) = \ell$. This function satisfies the following recurrence:

$$
Cost(v, \ell) = \begin{cases} 
0 & \text{if } v \text{ is a leaf} \\
\sum_{w \text{ child of } v} \min \{Cost(w, 2), Cost(w, 3)\} & \text{if } \ell = 1 \\
\sum_{w \text{ child of } v} \min \{Cost(w, 1) + 1, Cost(w, 3)\} & \text{if } \ell = 2 \\
\sum_{w \text{ child of } v} \min \{Cost(w, 1) + 1, Cost(w, 2) + 1\} & \text{if } \ell = 3 
\end{cases}
$$

Our task is to compute $\min\{Cost(\text{root}, 1), Cost(\text{root}, 2), Cost(\text{root}, 3)\}$.

We can memoize the $Cost$ function by associating three new fields $v.cost1$, $v.cost2$, and $v.cost3$ with each node $v$ in $T$.

Evaluating $Cost(v, \ell)$ at any node $v$ requires the values $Cost(w, \ast)$ at every child $w$ of $v$. Thus, it suffices to visit the nodes of $T$ using a postorder traversal.

To evaluate the subproblems at any node $v$, we spend constant time for each child of $v$. Equivalently, every node $w$ except the root contributes a constant amount of time to its parent. Thus, the memoized algorithm runs in $O(n)$ time.

```
MinCost(T):
    for each node $v$ of $T$ in postorder:
        $v.cost1 \leftarrow 0$
        $v.cost2 \leftarrow 0$
        $v.cost3 \leftarrow 0$
        for each child $w$ of $v$
            $v.cost1 \leftarrow v.cost1 + \min \{w.cost2, w.cost3\}$
            $v.cost2 \leftarrow v.cost2 + \min \{w.cost1 + 1, w.cost3\}$
            $v.cost3 \leftarrow v.cost3 + \min \{w.cost1, w.cost2\} + 1$
    return $\min\{T.cost1, T.cost2, T.cost3\}$
```

Rubric: Standard dynamic programming rubric. The postorder traversal can also be described recursively.