Activity Selection Problem

The Activity Selection Problem is a classic example where you're given a set of activities with their start and finish times, and the goal is to select the maximum number of activities that don't overlap. For this example, let's use a greedy approach, which is typically how the Activity Selection Problem is solved, noting that dynamic programming can be overkill for this specific problem due to its greedy-choice property. However, to fit the request, let's conceptualize a way it could be approached with dynamic programming for educational purposes.

Pseudocode (Greedy Approach):

```
Algorithm ActivitySelection(activities):
    Sort activities by finish times
    lastSelected = 0
    selectedActivities = [activities[0]]
    for i from 1 to length(activities) - 1:
        if activities[i].start >= activities[lastSelected].finish:
            selectedActivities.add(activities[i])
            lastSelected = i
    return selectedActivities
```

Dynamic Programming Approach Conceptualization:

For a dynamic programming approach to the Activity Selection Problem, one would typically consider the problem of selecting activities from `i` to `j` and finding the optimal solution for this subproblem, storing the results in a table.

Example Activities:

Let's consider a simplified example where each activity is represented by a start and finish time.

Activity	Start	Finish
A1	1	4
A2	3	5
A3	0	6
A4	5	7
A5	3	9
A6	5	9
A7	6	10
A8	8	11
A9	8	12
A10	2	14
A11	12	16

Steps for a Hypothetical DP Approach:

1. **Sort Activities**: Sort activities by finish time.

2. **Initialize Table**: Create a 2D table `dp`, where `dp[i][j]` represents the maximum number of activities that can be selected from activity `i` to activity `j`.
3. **Base Case**: If `i == j`, `dp[i][j] = 1` since only one activity is considered.
4. **Recursive Step**: For each pair `(i, j)`, consider all activities `k` where `i < k < j`. The value of `dp[i][j]` would be the maximum of `dp[i][k] + dp[k][j]` plus one for the `k`th activity if it fits.
5. **Fill the Table**: Start filling the table for all ranges, beginning from the shortest range to the longest.

Hypothetical DP Table (Partial and Conceptual):

Note: This table would be conceptual since the direct DP approach is not standard for this problem due to its efficiency being outmatched by the greedy method. However, we can imagine a table that tracks the maximum number of activities between any two activities based on start and finish times compatibility.

	A1	A2	A3	...	A11
A1	1				
A2		1			
A3			1		
...				...	
A11					1

- The cells would be filled based on compatibility, with non-directly adjacent cells considering whether including an activity would increase the total count without time overlap. (Cells filled diagonally)

Conclusion:

The activity selection problem is best solved with a greedy algorithm due to its nature of making a locally optimal choice at each step with a global optimum guarantee. The dynamic programming approach, while interesting to consider for educational purposes, is not the most efficient way to tackle this specific problem but offers a good exercise in understanding DP's applicability and limitations.

The Assembly Line Scheduling or Traversal Problem is a classic example that fits well into the dynamic programming paradigm. The problem involves finding the minimum time required to assemble a product when there are two assembly lines, each with its own set of stations. Each station has a different processing time, and there are additional times for transferring a product between the lines at each station.

Problem Setup:

- Two assembly lines, 1 and 2.
- `n` stations per assembly line, stations are numbered `1` to `n`.
- `a[i][j]`: Time taken at station `j` on line `i`.
- `t[i][j]`: Time to transfer from station `j` on line `i` to station `j+1` on the other line.
- `e[i]`: Entry time for line `i`.
- `x[i]`: Exit time for line `i`.

Objective:

Compute the minimum time required to assemble a product.

Pseudocode:

```
Algorithm AssemblyLineScheduling(a, t, e, x):
    Input: Arrays a, t, e, x as described above
    Output: Minimum time to assemble the product

    n = length(a[1])  // Number of stations
    F1 = new Array[n+1]
    F2 = new Array[n+1]

    // Base case: entry times
    F1[1] = e[1] + a[1][1]
    F2[1] = e[2] + a[2][1]

    // Build up the table F1 and F2 for each station
    for j from 2 to n:
        F1[j] = min(F1[j-1] + a[1][j], F2[j-1] + t[2][j-1] + a[1][j])
        F2[j] = min(F2[j-1] + a[2][j], F1[j-1] + t[1][j-1] + a[2][j])

    // Consider exit times
    return min(F1[n] + x[1], F2[n] + x[2])
```

Steps to Perform the Computation with a Table:

1. **Initialize**: Create two tables (arrays) `F1` and `F2` to store the minimum time to reach each station on line 1 and line 2, respectively.
2. **Fill Base Cases**: The first station's time is the entry time plus the processing time at the first station for both lines.
3. **Iterate and Fill**: For each subsequent station, calculate the minimum time considering the time to stay on the same line or switch lines from the previous station. Update `F1` and `F2` accordingly.
4. **Final Step**: The minimum time to assemble the product includes considering the exit time from the last station on both lines.

Example Table:

For simplicity, let's assume 3 stations, with made-up times.

Station	Line 1	Line 2
Entry	e[1]=2	e[2]=4
1	7	9
2	9	7
3	3	4
Exit	x[1]=3	x[2]=2

Assuming transfer times `t[1][j]` and `t[2][j]` are all 2 for all stations for simplicity.

Assembly Line Traversal

Filling Strategy:

- **Row Major**: Since the computation at each station depends on the previous one, we fill the table row by row, which corresponds to station by station in our context.
- The table is filled linearly from the start (after considering entry times) to the end (before considering exit times), reflecting the sequence of operations.

Actual Table After Computation:

Let's illustrate partial computation for clarity, focusing on filling strategy:

Station	F1	F2
1	9 (2+7)	13 (4+9)
2	16 (min(9+9, 13+2+7))	15 (min(13+7, 9+2+9))
3	19 (min(16+3, 15+2+4))	17 (min(15+4, 16+2+3))
Exit	22 (19+3)	19 (17+2)

Conclusion:

- The minimum assembly time is `min(22, 19) = 19`.
- The table filling is **Row Major** for this problem, as we progress through the stations linearly, updating the cumulative minimum time for reaching each station on both lines. This reflects the sequence of assembling operations and how decisions at each station affect subsequent choices.

Coin Changing Problem

The Coin Changing Problem is a classic problem in dynamic programming that asks: given an amount `N` and a set of coin denominations, what is the minimum number of coins needed to make change for `N`?

Problem Statement:

Given an amount `N` and a list of coin denominations `C = {C1, C2, ..., Cm}`, find the minimum number of coins needed to make change for `N`.

Pseudocode:

```
Algorithm CoinChange(N, C):
    Input: An amount N, and a list of coin denominations C[1...m]
    Output: Minimum number of coins needed to make change for N

    Let MinCoins[0...N] be an array where MinCoins[i] will store the minimum number of coins needed for
amount i
    Initialize MinCoins[0] = 0
    For i from 1 to N:
        MinCoins[i] = Infinity
        For j from 1 to m:
            If i >= C[j]:
                MinCoins[i] = min(MinCoins[i], MinCoins[i - C[j]] + 1)
    Return MinCoins[N]
```

Steps to Perform the Computation with a Table:

1. **Initialize the Table**: Create an array `MinCoins` of size `N+1` where each entry `MinCoins[i]` will store the minimum number of coins required to make change for amount `i`.
2. **Base Case**: Set `MinCoins[0] = 0` because the minimum number of coins needed to make change for 0 is 0.
3. **Filling the Table (Row Major)**:
    - Iterate through amounts from `1` to `N`, and for each amount, calculate the minimum number of coins required by considering all denominations.

    - For each denomination that does not exceed the current amount `i`, update `MinCoins[i]` to the minimum of its current value or `1 + MinCoins[i - denomination]`.

Filling Strategy:

- **Row Major**: The table is filled from left to right, updating `MinCoins` for each amount from `1` to `N` based on the available coin denominations.

Example with a Table:

Given:

- Amount `N = 11`
- Coin Denominations `C = {1, 5, 6}`

Initialization:

- `MinCoins[0...11] = {0, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞}`

Actual Table After Computation:

- **Note**: Table is conceptualized as an array here for simplicity.

Amount (i)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
MinCoins[i]	0	1	2	3	4	1	1	2	2	3	2	2

Explanation of Key Entries:

- `MinCoins[5] = 1`: Use one 5-coin.
- `MinCoins[6] = 1`: Use one 6-coin.
- `MinCoins[7] = 2`: Use one 6-coin and one 1-coin.
- `MinCoins[11] = 2`: Use one 5-coin and one 6-coin.

Conclusion:

The minimum number of coins needed to make change for 11 is 2. The table `MinCoins` is filled in **Row Major** order because we calculate the minimum coins needed for each amount starting from 1 up to N sequentially. This strategy ensures that when calculating `MinCoins[i]`, the minimum coins needed for all amounts less than `i` have already been determined.

Edit Distance

The Edit Distance (also known as the Levenshtein Distance) problem measures the minimum number of operations required to transform one string into another, where the operations include insertions, deletions, or substitutions of characters. This problem is a classic example of dynamic programming.

Problem Statement:

Given two strings `str1` and `str2`, find the minimum number of operations (insertions, deletions, substitutions) required to transform `str1` into `str2`.

Pseudocode:

```
Algorithm EditDistance(str1, str2):
    Input: Strings str1 and str2
    Output: Minimum number of operations to transform str1 into str2

    m = length(str1)
    n = length(str2)
    Let dp[m+1][n+1] be a table where dp[i][j] represents the edit distance between the first i characters
of str1 and the first j characters of str2

    // Initialize the table
    For i from 0 to m:
        dp[i][0] = i  // deletion
    For j from 0 to n:
        dp[0][j] = j  // insertion

    // Fill dp table
    For i from 1 to m:
        For j from 1 to n:
            If str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1]  // No operation needed
            Else:
                dp[i][j] = 1 + min(dp[i-1][j],    // deletion
                                   dp[i][j-1],    // insertion
                                   dp[i-1][j-1])  // substitution
    Return dp[m][n]
```

Steps to Perform the Computation with a Table:

1. **Initialize Table**: Create a 2D table `dp` with dimensions `(m+1) x (n+1)` where `m` and `n` are the lengths of `str1` and `str2`, respectively.
2. **Base Cases**: Initialize the first row and column of the table to represent the number of operations needed to convert a string of length `i` or `j` into an empty string (all insertions or deletions).
3. **Fill the Table (Row Major)**: Iterate through each cell of the table, filling in the minimum number of operations based on the operations performed on the previous parts of the strings.

Table Structure:

The table structure is a 2D matrix where rows represent characters (including an initial empty state) of `str1` and columns represent characters (including an initial empty state) of `str2`.

Filling Strategy:

- **Row Major**: This problem's table is filled in a row-major order because each cell's value depends on the operations performed on the substrings ending at the characters represented by the row and column indices.

Example:

Given `str1 = "kitten"` and `str2 = "sitting"`.

Initialization and Table:

```
Initial Table (with base cases filled):
    |   | - | s | i | t | t | i | n | g |
```

-	0	1	2	3	4	5	6	7
k	1							
i	2							
t	3							
t	4							
e	5							
n	6							

Final Filled Table (Partial):

	-	s	i	t	t	i	n	g
-	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

Conclusion:

- The minimum edit distance is found at `dp[m][n]`, which, in this case, is `dp[6][7] = 3`, indicating that 3 operations are required to transform "kitten" into "sitting".
- The table is filled in **Row Major** order, processing one row at a time, which sequentially builds up the solution based on previously computed values.

Longest Common Subsequence

The Longest Common Subsequence (LCS) problem is a classic example to illustrate dynamic programming. It involves finding the length of the longest subsequence present in both of two given sequences (strings or arrays). A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Pseudocode:

```
Algorithm LCS(X, Y):
    Input: Sequences X[1...m], Y[1...n]
    Output: Length of the longest common subsequence of X and Y

    Let L be a 2D array of dimensions (m+1) x (n+1)
    For i from 0 to m:
        For j from 0 to n:
            If i == 0 or j == 0:
                L[i][j] = 0
            Else if X[i] == Y[j]:
                L[i][j] = L[i-1][j-1] + 1
            Else:
                L[i][j] = max(L[i-1][j], L[i][j-1])
    Return L[m][n]
```

Steps to Perform the Computation with a Table:

1. **Initialize the Table**: Create a 2D array `L` with dimensions `(m+1) x (n+1)`, where `m` is the length of sequence `X`, and `n` is the length of sequence `Y`. The extra row and column account for the base case when one of the sequences is of length 0.
2. **Fill Base Cases**: Fill the first row and the first column of `L` with 0s, representing the scenarios when one of the sequences is empty.
3. **Iterative Filling**:
    - Fill the table row by row (or column by column), starting from `L[1][1]`.
    - If `X[i]` equals `Y[j]`, set `L[i][j] = L[i-1][j-1] + 1`.
    - If `X[i]` does not equal `Y[j]`, set `L[i][j] = max(L[i-1][j], L[i][j-1])`.

Table Structure:

- The table will be a 2D matrix of size `(m+1) x (n+1)`.

Filling Strategy:

- **Row Major**: This problem's table is filled in a row-major order, processing one row completely before moving to the next.

Example:

Given sequences `X = "AGGTAB"` and `Y = "GXTXAYB"`:

Initialization:

- `m = 6`, `n = 7`
- Table `L` is of dimensions `(7 x 8)`.

Actual Table After Computation:

```
    G X T X A Y B
  0 0 0 0 0 0 0 0
A 0 0 0 0 0 1 1 1
G 0 1 1 1 1 1 1 1
G 0 1 1 1 1 1 1 1
T 0 1 1 2 2 2 2 2
A 0 1 1 2 2 3 3 3
B 0 1 1 2 2 3 3 4
```

Explanation:

- The length of the longest common subsequence is found in `L[6][7]`, which is `4`. The LCS is `"GTAB"`.
- Each cell `L[i][j]` represents the length of the LCS of the prefixes `X[1...i]` and `Y[1...j]`.
- This approach ensures that all the required subproblems are solved systematically, leading to the final solution.

Conclusion:

The LCS problem demonstrates how dynamic programming can efficiently solve problems by breaking them down into smaller subproblems, solving each subproblem just once, and storing their solutions.

Fibonacci sequence

The Fibonacci sequence is a well-known example that's perfect for illustrating dynamic programming. The Fibonacci sequence is defined as follows: `Fib(0) = 0`, `Fib(1) = 1`, and `Fib(n) = Fib(n-1) + Fib(n-2)` for `n > 1`. Let's explore how to solve this using dynamic programming with table filling.

Pseudocode:

```
Algorithm Fibonacci(n):
    If n is 0 or 1:
        Return n
    Create an array fib[0...n]
    fib[0] = 0
    fib[1] = 1
    For i from 2 to n:
        fib[i] = fib[i-1] + fib[i-2]
    Return fib[n]
```

Steps to Perform the Computation with a Table:

1. **Initialize the Table**: Create an array `fib` with a length of `n+1`, where `n` is the Fibonacci number you want to calculate. The array index represents the number, and the value at each index will be the Fibonacci number corresponding to that index.
2. **Base Cases**: Fill in the base cases:
   - `fib[0] = 0`
   - `fib[1] = 1`
3. **Fill the Table (Row Major)**:
   - For each index `i` from `2` to `n`, compute `fib[i]` as the sum of the two preceding numbers (`fib[i-1]` and `fib[i-2]`).
   - Fill the table in a linear fashion, computing each Fibonacci number based on the previously computed values.

Table Structure:

- The table structure used here is a **1-dimensional array** because the Fibonacci sequence is a series of values based on the two preceding numbers. Each cell in the array corresponds to a Fibonacci number, with its index representing the step in the sequence.

Filling Strategy:

- **Row Major**: The table is filled in a linear (row major) fashion from left to right because each value depends only on the previous two values. This is a straightforward sequence where each entry is computed in order.

Example Table (Actual Table for `n=7`):

Let's simulate the filling of the table for `Fib(7)`:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
fib[]	0	1	1	2	3	5	8	13

Explanation of Key Entries:

- `fib[2] = fib[1] + fib[0] = 1 + 0 = 1`
- `fib[3] = fib[2] + fib[1] = 1 + 1 = 2`
- ...
- `fib[7] = fib[6] + fib[5] = 8 + 5 = 13`

Conclusion:

The minimum computation required to find `Fib(7)` is 13, which is found by filling the table in a row-major manner, ensuring that all dependencies are resolved sequentially. This dynamic programming approach eliminates the redundancy of recalculating the Fibonacci sequence for lower numbers multiple times, significantly reducing the computational cost.

The Chessboard Traversal for maximal profit problem can be framed as follows: Given a chessboard where each cell has a profit value associated with it, find the maximum profit you can accumulate by moving from the top-left corner (0,0) to the bottom-right corner (n-1,n-1), moving only to the right or down.

Pseudocode:

```
Algorithm ChessboardMaxProfit(P):
    Input: 2D array P representing the profit values of each cell on the chessboard
    Output: Maximum profit from top-left to bottom-right

    n = length(P)  // Assuming a square chessboard
    DP = 2D array of size n x n, initialized with 0s

    DP[0][0] = P[0][0]  // Starting point

    // Fill the first row and first column
    For i from 1 to n-1:
        DP[i][0] = DP[i-1][0] + P[i][0]  // First column
        DP[0][i] = DP[0][i-1] + P[0][i]  // First row

    // Fill the rest of the DP table
    For i from 1 to n-1:
        For j from 1 to n-1:
            DP[i][j] = max(DP[i-1][j], DP[i][j-1]) + P[i][j]

    Return DP[n-1][n-1]
```

Steps to Perform the Computation with a Table:

1. **Initialize the DP Table**: Create a 2D table `DP` with the same dimensions as the chessboard, where `DP[i][j]` represents the maximum profit that can be obtained upon reaching cell `(i,j)`.
2. **Base Case**: The profit at the starting cell `(0,0)` is the profit value of that cell itself.
3. **Fill First Row and Column**: Since you can only move right or down, the only way to reach any cell in the first row or column is from its immediate left or top cell, respectively.
4. **Fill the Rest of the Table**:
    - For each cell `(i,j)` (excluding the first row and column), calculate the maximum profit by taking the higher of two possible paths (from the left `DP[i][j-1]`, or from above `DP[i-1][j]`) and adding the current cell's profit `P[i][j]`.
5. **Table Filling Strategy**: **Row Major**, since each cell's maximum profit is calculated using values from the previous row or the current row's previous cells.

Table Structure:

- A 2D array `DP` with the same dimensions as the chessboard.

Example:

Given a 3x3 chessboard with the following profit values:

2	**3**	**0**
7	1	8
0	5	2

Actual Table After Computation:

	0	***1***	***2***
0	2	5	5
1	9	10	18
2	9	15	20

- The maximum profit to reach each cell is calculated based on the best path to get there, adding the cell's profit to the accumulated profit from the previous step.
- **Filling Method**: The table is filled in **Row Major** order. Each row is filled from left to right, starting from the top row and moving down to the bottom row.

Conclusion:

The maximal profit from traversing the chessboard from the top-left to the bottom-right, following the given rules, is 20. The DP table is filled in a row-major order, where the calculation for each cell is dependent on the maximum profit calculated for the cells immediately to its left and above it.