Photo by Cookie the Pom on Unsplash

Dynamic Programming (DP) is a powerful technique for solving optimization problems by breaking them down into overlapping subproblems.

I. TIPS & STRATEGIES FOR SOLVING DYNAMIC PROGRAMMING PROBLEMS:

1. Understand the Problem:

- Clearly understand the problem statement, constraints, and the optimization goal.

1.1. Indicators of Dynamic Programming Problems:

i. Optimal Substructure:

- The problem can be broken down into smaller, overlapping subproblems that can be solved independently.
- The optimal solution to the problem can be constructed from the optimal solutions of its subproblems.

ii. Overlapping Subproblems:

- The same subproblems are solved multiple times in the process of solving the overall problem.
- Repeated computation of the same subproblems can be avoided by using memoization or tabulation.

iii. Recursive Structure:

- The problem can be formulated in a recursive manner, where the solution to the problem depends on the solutions to smaller instances of the same problem.

iv. Memoization or Tabulation:

- If the problem can be solved using memoization (caching the results of recursive calls) or tabulation (building up the solution iteratively), it may be a good candidate for dynamic programming.

v. Dependency on Previous Steps:

- The solution to the problem depends on the solutions to some of its previous steps or stages.

vi. Sequential Decision Making:

- If the problem involves making a sequence of decisions and the optimal decision at each step depends on the decisions made in previous steps, dynamic programming may be applicable.

vii. Top-Down vs. Bottom-Up Approach:

- If you can solve the problem using a top-down approach (recursion with memoization) or a bottom-up approach (iterative approach with tabulation), it suggests a dynamic programming solution.

viii. Common DP Keywords:

- Look for keywords in the problem statement like “minimum,” “maximum,” “shortest,” “longest,” “count,” etc. These words often

indicate optimization problems that can be solved using dynamic programming.

ix. Examples of DP Problems:

- Problems involving finding the shortest/longest path, maximizing or minimizing a value, or counting the number of ways to do something are often solved using dynamic programming.

1.2. Key Points for Understanding and Solving Dynamic Programming Problems:

i. Understand the Problem:

- Carefully read and understand the problem statement.
- Identify the input and output requirements.
- Recognize the optimization goal (minimization or maximization).

ii. Identify Overlapping Subproblems:

- Look for repeating subproblems within the larger problem.
- Recognize that the same subproblem is solved multiple times.

iii. Define Optimal Substructure:

- Confirm that the optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems.
- Break down the problem into smaller, manageable subproblems.

iv. Recurrence Relation:

- Formulate a recurrence relation that expresses the solution to the problem in terms of solutions to smaller instances of the same problem.
- Define the relationship between the current state and its subproblems.

v. Memoization (Top-Down):

- Use memoization to cache the results of previously solved subproblems.
- Avoid redundant computations by checking the memoization table before solving a subproblem.

vi. Tabulation (Bottom-Up):

- Start solving the problem from the smallest subproblems and build up to the overall problem.
- Use a table or array to store intermediate results.

vii. Base Cases:

- Clearly define the base cases that represent the smallest subproblems.
- Ensure that the base cases are simple to solve directly.

viii. Optimize Space Complexity:

- Optimize space usage by considering whether you need to store all intermediate results or only a subset of them.
- Often, you can reduce the space complexity by using rolling arrays or other memory optimization techniques.

ix. Iterate and Refine:

- Iterate on your solution and refine the recurrence relation.
- Optimize the algorithm based on patterns you observe.
- Experiment with different approaches to improve efficiency.

x. Think in Terms of States:

- Identify the state variables that uniquely define a subproblem.
- Understand how the values of these variables change as you move from one subproblem to the next.

xi. Check for Constraints:

- Be mindful of the time and space constraints, and ensure your solution meets the requirements.

xii. Debugging Memoization Errors:

- If using memoization, ensure that you correctly handle memoization table updates and that you don’t miss any cases.

- Debug by printing intermediate results and tracing the function calls.

2. Identify Overlapping Subproblems:

- Recognize patterns of overlapping subproblems. DP problems often involve solving the same subproblems multiple times.

```
# Example that demonstrates a recursive solution with overlapping subproblems
# Problem: Fibonacci sequence

# Naive recursive solution without memoization
def fibonacci_naive(n):
    if n <= 1:
        return n
    else:
        return fibonacci_naive(n - 1) + fibonacci_naive(n - 2)

# Test the naive implementation
print(f"Fibonacci of 5 (naive): {fibonacci_naive(5)}")
```

- The `fibonacci_naive` function calculates Fibonacci numbers using a recursive approach. However, it suffers from overlapping subproblems because it repeatedly calculates the Fibonacci of the same numbers multiple times, resulting in exponential time complexity.
- Now, let's enhance this with memoization to address the overlapping subproblems:

```
# Improved solution with memoization
def fibonacci_memoization(n, memo={}):
    if n <= 1:
        return n

    # Check if the result is already in the memoization table
    if n not in memo:
        # If not, calculate the result and store it in the memoization table
        memo[n] = fibonacci_memoization(n - 1, memo) + fibonacci_memoization(n -

    return memo[n]

# Test the memoized implementation
print(f"Fibonacci of 5 (memoized): {fibonacci_memoization(5)}")
```

- In the `fibonacci_memoization` function, we use a dictionary (`memo`) to store the results of previously solved subproblems.

- Before computing the Fibonacci of a number, we check if it's already in the memoization table.
- If it is, we return the precomputed result; otherwise, we calculate it and store the result in the table.

This memoized version significantly improves the time complexity by avoiding redundant calculations, demonstrating the concept of overcoming overlapping subproblems in dynamic programming.

3. Define State:

- Clearly define the state of the problem. Identify the variables that uniquely represent the problem's state.
- The state of a dynamic programming problem is a set of parameters or variables that uniquely represent a specific subproblem. These variables capture the essential information needed to solve that subproblem. By defining the state, you create a clear understanding of the problem's structure and pave the way for building a dynamic programming solution.

3.1. Guidline for Defining a State:

i. Identify Relevant Variables:

- Consider the problem's requirements and constraints.
- Identify the variables that play a role in determining the solution.

ii. Capture Essential Information:

- The state should contain the minimum information required to compute the solution to a subproblem.
- It should exclude redundant or unnecessary information.

iii. Make States Independent:

- Ensure that each subproblem's state is independent, meaning the solution to one subproblem does not depend on the solution to another with the same state.

iv. Consider Recurrence Relation:

- Think about how the variables in the state relate to each other and how they contribute to the recurrence relation.
- The recurrence relation is the mathematical relationship between the current state and its subproblems.

v. Base Cases:

- The state definition should be consistent with the base cases of the problem.
- Base cases represent the smallest subproblems and should be easily solvable.

```
# Define state for Fibonacci problem

def fibonacci_dp(n):
    # State: The current position 'n'
    # Variables: 'n'

    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1

    # Recursive case
    # Recurrence relation: F(n) = F(n-1) + F(n-2)
    return fibonacci_dp(n - 1) + fibonacci_dp(n - 2)

# Test the Fibonacci dynamic programming solution
print(f"Fibonacci of 5 (DP): {fibonacci_dp(5)}")
```

- The state is defined by the variable '`n`', representing the position in the Fibonacci sequence.
- The state captures the essential information needed to compute the solution to the Fibonacci subproblem for a given position.
- This state is used to construct the recurrence relation and solve the problem efficiently using dynamic programming techniques.

4. Recurrence Relation:

- Formulate a recurrence relation that expresses the optimal solution in terms of solutions to smaller subproblems. This is often the most crucial step in DP.
- It defines the relationship between the optimal solution to the original problem and the solutions to its smaller subproblems.

- The recurrence relation provides a way to express the problem recursively, breaking it down into smaller, manageable parts.

4.1. Guideline for Formulating a Recurrence Relation:

i. Identify Subproblems:

- Break down the problem into smaller subproblems that share the same structure.
- Consider how the solution to the original problem can be constructed from the solutions to these subproblems.

ii. Define the Recurrence Relation:

- Express the solution to the current problem in terms of solutions to smaller instances of the same problem.
- Use mathematical notation to represent the relationship.

iii. Base Cases:

- Clearly define the base cases where the solution is known without further recursion.
- Base cases represent the smallest subproblems and are essential for terminating the recursion.

iv. Consider State Variables:

- Refer to the state variables defined earlier, as they play a key role in the recurrence relation.
- Express the state transition in terms of these variables.

v. Ensure Overlapping Subproblems:

- The recurrence relation should involve overlapping subproblems, reinforcing the idea that solutions to the same subproblem are reused.

```
# Formulate the recurrence relation for the Fibonacci problem
def fibonacci_dp(n):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
```

```
            return 1

        # Recurrence relation: F(n) = F(n-1) + F(n-2)
        return fibonacci_dp(n - 1) + fibonacci_dp(n - 2)

# Test the Fibonacci dynamic programming solution
print(f"Fibonacci of 5 (Recurrence): {fibonacci_dp(5)}")
```

- The recurrence relation is explicitly expressed as `F(n) = F(n-1) + F(n-2)`, where `F(n)` represents the Fibonacci number at position 'n'.
- This relation reflects the recursive nature of the Fibonacci sequence, where the solution to the current position depends on the solutions to the two preceding positions.
- Implementing this relation efficiently with memoization or tabulation leads to a dynamic programming solution for the Fibonacci problem.

5. Base Cases:

- Define the base cases that represent the smallest subproblems. Ensure that the base cases are easy to solve directly.

5.1. Guideline for Defining Base Cases:

i. Identify Smallest Subproblems:

- Consider what constitutes the smallest instances of the problem.
- Identify situations where the solution is known without further recursion.

ii. Make Base Cases Explicit:

- Clearly state the conditions that trigger the base cases.
- Make the base cases explicit in your code.

iii. Ensure Easy Solvability:

- Ensure that the solutions to the base cases are straightforward and can be computed directly without further recursion.

iv. Provide Initial Values:

- For problems with memoization or tabulation, set initial values for the base cases in the data structures used for storing intermediate results.

```
# Define base cases for the Fibonacci problem
def fibonacci_dp(n, memo={}):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1

    # Check if the result is already in the memoization table
    if n not in memo:
        # If not, calculate the result and store it in the memoization table
        memo[n] = fibonacci_dp(n - 1, memo) + fibonacci_dp(n - 2, memo)

    return memo[n]

# Test the Fibonacci dynamic programming solution with base cases
print(f"Fibonacci of 5 (Base Cases): {fibonacci_dp(5)}")
```

- The base cases are explicitly defined as `n == 0` and `n == 1`, where the Fibonacci numbers are 0 and 1, respectively.
- These base cases represent the smallest subproblems, and their solutions are known without further recursion.
- By handling these cases explicitly, we ensure that the dynamic programming algorithm can terminate when it reaches the base cases and then build up the solution from there.

6. Memoization (Top-Down):

- Implement the top-down approach using memoization. Store the results of subproblems in a data structure (usually a table or a dictionary) to avoid redundant calculations.

```
# Top-down approach with memoization for the Fibonacci problem
def fibonacci_memoization(n, memo={}):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1

    # Check if the result is already in the memoization table
    if n not in memo:
        # If not, calculate the result and store it in the memoization table
        memo[n] = fibonacci_memoization(n - 1, memo) + fibonacci_memoization(n -

    return memo[n]

# Test the Fibonacci top-down solution with memoization
print(f"Fibonacci of 5 (Top-Down Memoization): {fibonacci_memoization(5)}")
```

- The `memo` dictionary is used to store the results of subproblems. The keys are the values of the state variable `n`, and the values are the corresponding Fibonacci numbers.
- Before calculating the Fibonacci number for a particular `n`, the algorithm checks whether the result is already in the memoization table. If it is, the precomputed result is returned, avoiding redundant calculations.
- If the result is not in the memoization table, the algorithm calculates it using recursion and stores the result in the memoization table before returning it.

This top-down approach with memoization helps optimize the time complexity by avoiding repeated computations, making the solution more efficient compared to the naive recursive approach.

7. Tabulation (Bottom-Up):

- Implement the bottom-up approach using tabulation. Start solving the problem from the smallest subproblems and build up to the optimal solution.
- This approach avoids recursion and starts with the smallest subproblems, gradually solving larger subproblems until we arrive at the optimal solution.

```
# Bottom-up approach with tabulation for the Fibonacci problem
def fibonacci_tabulation(n):
    # Initialize an array to store the Fibonacci numbers
    fib_table = [0] * (n + 1)

    # Base cases
    fib_table[0] = 0
    fib_table[1] = 1

    # Fill in the table from the bottom up
    for i in range(2, n + 1):
        fib_table[i] = fib_table[i - 1] + fib_table[i - 2]

    # The result is in the last element of the table
    return fib_table[n]

# Test the Fibonacci bottom-up solution with tabulation
print(f"Fibonacci of 5 (Bottom-Up Tabulation): {fibonacci_tabulation(5)}")
```

- We use an array (`fib_table`) to store the results of subproblems. The index of the array corresponds to the value of the state variable `n`, and

the array elements store the corresponding Fibonacci numbers.

- The base cases are initialized directly in the table (`fib_table[0]` and `fib_table[1]`).
- We iterate from the smallest subproblems (index 2) up to the target value of `n` , filling in the table iteratively using the recurrence relation `fib_table[i] = fib_table[i - 1] + fib_table[i - 2]` .
- The final result is obtained from the last element of the table (`fib_table[n]`).

The bottom-up approach with tabulation is often more space-efficient than the top-down approach with memoization since it avoids the overhead of function calls and can be implemented using a simple array. It's particularly useful when we only need to calculate the final result and don't need to keep track of intermediate values.

8. State Transition Matrix:

- For problems with multiple variables in the state, visualize a state transition matrix to understand the relationships between different states.
- Each entry in the matrix represents the transition from one state to another and may contain information about the optimal solution or the cost associated with transitioning between states. This matrix helps in understanding the dependencies and relationships among various subproblems.

```
# Example: Coin change
# The goal is to find the minimum number of coins needed to make up a given amou

def coinChange(coins, amount):
    # Initialize a matrix to represent state transitions
    # Rows represent different amounts, columns represent different coin denomin
    dp = [[float('inf')] * (len(coins) + 1) for _ in range(amount + 1)]

    # Base case: The minimum number of coins needed to make 0 is 0
    for j in range(len(coins) + 1):
        dp[0][j] = 0

    # Fill in the matrix using a bottom-up approach
    for i in range(1, amount + 1):
        for j in range(1, len(coins) + 1):
            # If the current coin denomination is greater than the amount, use t
            if coins[j - 1] > i:
                dp[i][j] = dp[i][j - 1]
            else:
                # Transition: Minimum of not using the current coin and using th
                dp[i][j] = min(dp[i][j - 1], 1 + dp[i - coins[j - 1]][j])
```

```
    # The final result is in the bottom-right corner of the matrix
    return dp[amount][len(coins)]

# Example usage
coins = [1, 2, 5]
amount = 11
result = coinChange(coins, amount)
print(f"Minimum number of coins needed: {result}")
```

- The state is represented by two variables: the remaining amount to be achieved (`amount`) and the available coin denominations (`coins`). We can use a state transition matrix to depict the relationships between different states.
- The state transition matrix `dp` is a 2D array where each entry `dp[i][j]` represents the minimum number of coins needed to make up amount `i` using the first `j` coin denominations. The transition is based on whether or not the current coin is used in the solution.

9. Optimize Space Complexity:

- If space complexity is a concern, optimize the DP solution to use only the necessary information for each subproblem. Sometimes, you only need information from the previous two states.
- This is often referred to as rolling arrays or constant space optimization.

```
# Modify the previous example of finding the length of the longest common subseq
# to use only two rows of the matrix, reducing space complexit

def longestCommonSubsequence(str1, str2):
    m, n = len(str1), len(str2)

    # Use two rows for space optimization
    dp = [[0] * (n + 1) for _ in range(2)]

    # Fill in the matrix using a bottom-up approach
    for i in range(1, m + 1):
        # Alternate between rows to represent the current and previous rows
        current_row, prev_row = i % 2, (i - 1) % 2
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:
                dp[current_row][j] = dp[prev_row][j - 1] + 1
            else:
                dp[current_row][j] = max(dp[prev_row][j], dp[current_row][j - 1]

    # Return the length of the LCS
    return dp[m % 2][n]

# Example usage
str1 = "ABCD"
str2 = "ACDF"
```

```
result = longestCommonSubsequence(str1, str2)
print(f"Length of Longest Common Subsequence: {result}")
```

- We use only two rows (`dp = [[0] * (n + 1) for _ in range(2)]`) instead of the full matrix to represent the current and previous rows.
- We alternate between the current and previous rows using the variables `current_row` and `prev_row`.
- The space optimization reduces the space complexity from O(m * n) to O(min(m, n)), where m and n are the lengths of the input strings.

This optimization is applicable when each state only depends on the information from the two previous states. It is a common technique to reduce the space requirements of dynamic programming solutions.

10. Optimal Substructure:

- Ensure that the problem exhibits the optimal substructure property, meaning that the optimal solution to the overall problem can be constructed from optimal solutions to its subproblems.
- This property allows dynamic programming algorithms to break down a complex problem into simpler, overlapping subproblems, solving each subproblem only once and reusing its solution when needed.

10.1. Key Aspects to Ensure Optimal Substructure:

i. Define the Problem in Terms of Subproblems:

- Express the original problem as a combination of smaller, overlapping subproblems.
- Identify how the optimal solution to the larger problem depends on the optimal solutions to its subproblems.

ii. Recurrence Relation:

- Formulate a recurrence relation that relates the solution to the current problem to the solutions of its subproblems.
- The recurrence relation should capture the optimal choice at each step and how it contributes to the overall optimal solution.

iii. Base Cases:

- Define base cases that represent the simplest subproblems and provide direct solutions.
- Ensure that the base cases align with the problem's definition and are used to terminate the recursion.

iv. Ensure Independence of Subproblems:

- Confirm that the solution to one subproblem is independent of the solutions to other subproblems.
- The independence allows for efficient computation and reuse of solutions without redundancy.

v. Demonstrate Constructibility:

- Show that the optimal solution to the overall problem can be constructed by combining optimal solutions to its subproblems.
- This is often demonstrated through a step-by-step construction or by showing that the choices made at each step contribute to the overall optimality.

vi. Avoid Circular Dependencies:

- Ensure that there are no circular dependencies among subproblems, meaning that a subproblem's solution doesn't depend on the solution of a subproblem that, in turn, depends on the first subproblem.

vii. Iterative vs. Recursive Formulation:

- Both iterative (bottom-up) and recursive (top-down) formulations should exhibit optimal substructure.
- The choice between the two depends on the problem and personal preference.

```
# Example: Shortest Path Problem
def shortest_path(graph, start, end):
    # Base case: If start and end are the same, the shortest path is the empty p
    if start == end:
        return 0

    # Recursive case: Find the shortest path by considering all possible interme
    min_path = float('inf')
    for neighbor in graph[start]:
```

```
            path_cost = shortest_path(graph, neighbor, end) + graph[start][neighbor]
            min_path = min(min_path, path_cost)

        return min_path

# Example graph representation (weighted adjacency list)
graph = {
    'A': {'B': 2, 'C': 4},
    'B': {'C': 1, 'D': 7},
    'C': {'D': 3},
    'D': {}
}

# Test the shortest path function
start_vertex = 'A'
end_vertex = 'D'
result = shortest_path(graph, start_vertex, end_vertex)
print(f"Shortest Path from {start_vertex} to {end_vertex}: {result}")
```

- The optimal substructure is demonstrated by recursively finding the shortest path from the start vertex to the end vertex by considering all possible intermediate vertices. The overall shortest path can be constructed by combining the shortest paths to intermediate vertices.

11. Iterate Over Choices:

- For problems involving making choices (e.g., selecting or excluding items), iterate over all possible choices to find the optimal one.

11.1. Key Steps of Iteration Over Choices:

i. Identify Decision Points:

- Determine where in the problem-solving process you need to make decisions or choices.
- These decision points often correspond to selecting or excluding items, determining the order of operations, or making other strategic choices.

ii. Enumerate Possible Choices:

- List all possible choices that can be made at each decision point.
- This includes considering different options, such as selecting an item or excluding it, choosing an order of operations, or making other relevant decisions.

iii. Iterate Over Choices:

- Use loops or recursion to iterate over all possible choices.

- For each choice, evaluate its impact on the overall solution and consider the subproblem that results from making that choice.

iv. Evaluate Optimal Choice:

- Assess the impact of each choice on the overall objective function or criteria.
- Determine the optimal choice that leads to the best possible solution based on the subproblem solutions.

v. Update Memoization or Tabulation Table:

- If using memoization or tabulation, update the table with the results of each choice to avoid redundant computations.

```
def knapsack(values, weights, capacity, n, memo={}):
    # Base case: If no items or no capacity, return 0
    if n == 0 or capacity == 0:
        return 0

    # Check if the result is already in the memoization table
    if (n, capacity) in memo:
        return memo[(n, capacity)]

    # If the current item's weight exceeds the remaining capacity, exclude it
    if weights[n - 1] > capacity:
        result = knapsack(values, weights, capacity, n - 1, memo)
    else:
        # Consider both including and excluding the current item, and choose the
        include_current = values[n - 1] + knapsack(values, weights, capacity - w
        exclude_current = knapsack(values, weights, capacity, n - 1, memo)
        result = max(include_current, exclude_current)

    # Memoize the result before returning
    memo[(n, capacity)] = result
    return result

# Example usage
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
num_items = len(values)
result = knapsack(values, weights, capacity, num_items)
print(f"Maximum value in the knapsack: {result}")
```

- The function `knapsack` recursively explores all choices of including or excluding each item in the knapsack, updating the memoization table to avoid redundant computations.

- This iterative process allows the algorithm to find the optimal combination of items that maximizes the total value within the given capacity.

12. Think Backwards:

- When defining the recurrence relation, think backward from the end goal to understand how the solution can be built up step by step.
- By starting from the end goal and working backward, you can gain insights into how the optimal solution can be built up step by step from smaller subproblems. This helps in formulating a recurrence relation that captures the recursive nature of the problem.

12.1. Key Steps for Thinking Backwards:

i. Identify the End Goal:

- Clearly define the end goal or the final state that you want to achieve.
- Understand what the optimal solution to the overall problem looks like.

ii. Break Down into Subproblems:

- Consider how the problem can be broken down into smaller subproblems.
- Identify the variables or parameters that represent the state of these subproblems.

iii. Define Base Cases:

- Clearly define the base cases that represent the smallest subproblems or the stopping condition for the recursion.
- Ensure that the base cases are easy to solve directly.

iv. Formulate Recurrence Relation:

- Think about how the solution to the overall problem can be constructed from the solutions to its subproblems.
- Express this relationship in the form of a recurrence relation, capturing the recursive nature of the problem.

v. Consider Decision Points:

- If the problem involves making decisions at various points, consider how these decisions affect the overall solution.
- Think about the choices that lead to the optimal solution.

vi. Optimal Choice at Each Step:

- Determine the optimal choice at each step by considering the impact on the end goal.
- Identify the decisions or choices that contribute to the overall optimality.

```
# Example: the Longest Increasing Subsequence (LIS) problem

def lengthOfLIS(nums):
    if not nums:
        return 0

    n = len(nums)

    # Initialize an array to store the length of the LIS ending at each index
    lis = [1] * n

    # Iterate backward to fill in the LIS array
    for i in range(n - 2, -1, -1):
        for j in range(i + 1, n):
            if nums[i] < nums[j]:
                # If the current element is less than the next one, update the L
                lis[i] = max(lis[i], 1 + lis[j])

    # The maximum value in the LIS array is the length of the overall LIS
    return max(lis)

# Example usage
nums = [10, 9, 2, 5, 3, 7, 101, 18]
result = lengthOfLIS(nums)
print(f"Length of Longest Increasing Subsequence: {result}")
```

- The `lengthOfLIS` function uses a dynamic programming approach to find the length of the Longest Increasing Subsequence.
- It iterates backward through the array and updates the LIS array based on the choices that lead to the optimal solution.
- By thinking backward, we can efficiently determine the length of the LIS for the entire array.

13. Handle Constraints Efficiently:

- If the problem has constraints, find ways to incorporate them into the DP solution efficiently. This might involve modifying the state definition or

introducing additional variables.

13.1. Key Steps for Handling Constraints Efficiently:

i. Identify Constraints:

- Clearly understand the constraints imposed by the problem.
- Identify the aspects of the state or the solution that must adhere to these constraints.

ii. Modify State Definition:

- Adjust the state definition to include variables that capture the constraints.
- Ensure that the state fully represents the problem, considering both the optimization criteria and the constraints.

iii. Introduce Additional Variables:

- If necessary, introduce additional variables to keep track of information related to the constraints.
- These variables can be used in the recurrence relation to enforce and satisfy the constraints.

iv. Update Recurrence Relation:

- Modify the recurrence relation to account for the constraints.
- Consider how the constraints affect the transition from one state to another and incorporate these considerations into the recurrence relation.

v. Base Cases and Initialization:

- Ensure that the base cases and initialization steps consider the constraints.
- Initialize the memoization or tabulation structures in a way that satisfies the constraints.

```
# Example with constraints: the 0/1 Knapsack problem with an additional constrai
# We want to maximize the value of items selected while ensuring that the total
```

```
def knapsack_with_constraints(values, weights, capacity, max_weight):
    n = len(values)

    # Initialize a 3D array to represent state transitions
    dp = [[[0] * (max_weight + 1) for _ in range(capacity + 1)] for _ in range(n

    # Fill in the matrix using a bottom-up approach
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            for mw in range(max_weight + 1):
                # If the current item's weight exceeds the remaining capacity or
                if weights[i - 1] > w or weights[i - 1] > mw:
                    dp[i][w][mw] = dp[i - 1][w][mw]
                else:
                    # Consider both including and excluding the current item, an
                    include_current = values[i - 1] + dp[i - 1][w - weights[i -
                    exclude_current = dp[i - 1][w][mw]
                    dp[i][w][mw] = max(include_current, exclude_current)

    # The final result is in the bottom-right corner of the matrix
    return dp[n][capacity][max_weight]

# Example usage
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
max_weight_constraint = 25
result = knapsack_with_constraints(values, weights, capacity, max_weight_constra
print(f"Maximum value in the knapsack with constraints: {result}")
```

- The `knapsack_with_constraints` function handles both capacity and max weight constraints efficiently by extending the state definition and introducing an additional variable (`mw`) into the 3D array.
- This modification ensures that the constraints are incorporated into the dynamic programming solution.

14. Mathematical Optimization:

- Some problems can be mathematically optimized before applying DP. Look for mathematical properties that can simplify the solution.
- By identifying mathematical properties or relationships inherent in the problem, you can often derive closed-form solutions, recurrence relations, or constraints that significantly streamline the dynamic programming approach.

14.1. Key Steps for Leveraging Mathematical Optimization in Combination with Dynamic Programming:

i. Identify Mathematical Properties:

- Analyze the problem to identify any mathematical properties or relationships inherent in the problem statement.

- Look for patterns, symmetries, or constraints that can be exploited to simplify the solution.

ii. Explore Closed-Form Solutions:

- Investigate whether there are closed-form solutions or mathematical expressions that directly solve parts of the problem.
- Closed-form solutions can be more efficient than dynamic programming in some cases.

iii. Derive Recurrence Relations

- If a closed-form solution is not feasible, look for mathematical relationships that can be expressed as recurrence relations.
- Derive formulas or equations that describe how the solution to the problem depends on the solutions to smaller instances of the same problem.

iv. Simplify the State Space:

- Use mathematical insights to simplify the state space or state representation.
- Eliminate redundant states or parameters by taking advantage of mathematical properties.

v. Optimize Time Complexity:

- Mathematical optimization can sometimes lead to more efficient algorithms with better time complexity.
- Leverage mathematical properties to reduce the number of computations required to solve the problem.

vi. Consider Special Cases:

- Explore special cases or scenarios where mathematical properties can be exploited to find optimal solutions more efficiently.
- Special cases may allow for more direct and simplified solutions.

```
# Suppose we want to find the nth Fibonacci number using dynamic programming.
# Instead of using a standard dynamic programming approach, we can leverage the

import math

def fibonacci_binet(n):
    sqrt_5 = math.sqrt(5)
    phi = (1 + sqrt_5) / 2
    psi = (1 - sqrt_5) / 2

    # Binet's Formula for Fibonacci numbers
    fib_n = (phi**n - psi**n) / sqrt_5
    return round(fib_n)

# Example usage
n = 5
result = fibonacci_binet(n)
print(f"Fibonacci number at position {n}: {result}")
```

- Binet's Formula allows us to directly compute the nth Fibonacci number without the need for dynamic programming.
- While this formula is specific to Fibonacci numbers, similar mathematical optimizations can be explored for other problems to simplify the solution process before resorting to dynamic programming.

15. Reuse Previous Results:

- Leverage previously computed results whenever possible. DP is about reusing solutions to subproblems to avoid redundant computations.
- Dynamic programming aims to solve a problem by breaking it down into smaller overlapping subproblems and then reusing the solutions to those subproblems to avoid redundant computations.
- Example: in the Fibonacci problem, the `memo` dictionary is used to store and retrieve previously computed results for Fibonacci numbers. The function checks whether the result for a particular `n` is already in the memoization table before computing it, ensuring that solutions to subproblems are reused. This approach optimizes the time complexity by avoiding redundant computations.

15.1. Key Steps to Effectively Reuse Previous Results in Dynamic Programming:

i. Memoization (Top-Down):

- Use memoization to store and retrieve the results of previously solved subproblems.

- Before solving a subproblem, check if its solution is already stored in a memoization table (dictionary or array).

ii. Tabulation (Bottom-Up):

- Use tabulation to iteratively fill in a table (array) of solutions to subproblems from the smallest to the target subproblem.
- Ensure that each entry in the table is computed only once and relies on previously computed results.

iii. Define State Clearly:

- Clearly define the state of the problem, including the variables that uniquely represent the subproblems.
- Ensure that the state definition is sufficient for indexing memoization tables or tabulation arrays.

iv. Recompute Only When Necessary:

- When solving a subproblem, recompute its solution only if it has not been computed before.
- Use memoization to store and retrieve intermediate results efficiently.

v. Update Tables Incrementally:

- When using tabulation, update the table incrementally from the smallest subproblems to the target subproblem.
- Ensure that each entry in the table depends only on previously computed results.

vi. Optimize Space Complexity:

- Optimize space complexity by storing only the necessary information in memoization tables or tabulation arrays.
- Avoid unnecessary duplication of data and store only what is needed to compute the solution.

16. Use Bitmasks (for Subset Problems):

- If the problem involves subsets, consider using bitmasks to represent subsets efficiently and iterate through all possible subsets.
- Bitmasks use binary representation to encode the presence or absence of elements in a set. This approach is particularly useful for problems where you need to consider all possible subsets of a set of elements.

16.1. Key Steps to Use Bitmasks in Subset Problems:

i. Bitmask Representation:

- Use an integer variable (bitmask) where each bit represents the presence or absence of an element in the subset.
- If the ith bit is set to 1, it indicates that the ith element is included in the subset.

ii. Generating All Subsets:

- Iterate through all possible values of the bitmask to generate all subsets.
- The total number of possible subsets for a set of n elements is 2^n, so iterate from 0 to 2^n-1.

iii. Check Subset Elements:

- To check whether the ith element is included in the subset, use bitwise operations.
- For example, to check if the ith bit is set: `(bitmask & (1 << i)) != 0`.

iv. Subset Iteration Example:

- Iterate through all subsets using a bitmask and perform dynamic programming computations for each subset.

```
# Example: Subset Sum problem.
# Given a set of positive integers and a target sum, the task is to find whether

def subset_sum(nums, target_sum):
    n = len(nums)

    # Iterate through all possible subsets using bitmasks
    for bitmask in range(1 << n):
        current_sum = 0

        # Check each element in the subset
        for i in range(n):
            if bitmask & (1 << i) != 0:
```

```
                current_sum += nums[i]

        # Check if the current subset sums to the target sum
        if current_sum == target_sum:
            return True

    return False

# Example usage
nums = [3, 34, 4, 12, 5, 2]
target_sum = 9
result = subset_sum(nums, target_sum)
print(f"Subset with sum {target_sum} exists: {result}")
```

- The function `subset_sum` iterates through all possible subsets using a bitmask and checks whether the current subset's elements sum to the target sum.
- While this approach is not the most efficient for the Subset Sum problem, it demonstrates the use of bitmasks for subset problems.
- More sophisticated dynamic programming techniques can be applied to solve Subset Sum more efficiently using bitmasks and memoization.

17. Divide and Conquer:

- If a DP problem seems complex, break it down into smaller subproblems and solve each subproblem independently. This is especially useful for problems with a recursive structure.

18. Study Examples:

- Study solved examples to understand how DP is applied to different types of problems. Pay attention to the thought process behind defining states and recurrence relations.

II. COMMON PROBLEM TYPES:

1. Classic Dynamic Programming:

- Problem Type: Solve a problem by breaking it into smaller subproblems and solving each subproblem only once.

```
# Example Problem: “Climbing Stairs”
# You are climbing a staircase. It takes n steps to reach the top. Each time you
# In how many distinct ways can you climb to the top?

def climbStairs(n):
    # Base cases: 0 steps (1 way), 1 step (1 way), and 2 steps (2 ways)
```

```
    if n == 0:
        return 1
    elif n == 1:
        return 1
    elif n == 2:
        return 2

    # Initialize an array to store the number of ways to reach each step
    dp = [0] * (n + 1)

    # Base cases for 0, 1, and 2 steps
    dp[0] = 1
    dp[1] = 1
    dp[2] = 2

    # Fill in the array using a bottom-up approach
    for i in range(3, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    # The final result is the number of ways to reach the top step
    return dp[n]

# Example usage
n = 5
ways = climbStairs(n)
print(f"Distinct ways to climb {n} steps: {ways}")
```

- Define the `climbStairs` function, which takes an integer `n` representing the number of steps in the staircase.
- We handle the base cases where `n` is 0, 1, or 2.
- If there are 0 steps, there's only 1 way (doing nothing). If there's 1 step, there's only 1 way (climbing 1 step).
- If there are 2 steps, there are 2 ways (climbing 2 steps at once or climbing two 1-step increments).
- Create an array `dp` of length `n + 1` to store the number of ways to reach each step. This array will be used for dynamic programming, where each entry `dp[i]` represents the number of ways to reach the `i`-th step.
- Set the initial values for the base cases in the `dp` array. These values correspond to the base cases discussed earlier: 1 way for 0 steps, 1 way for 1 step, and 2 ways for 2 steps.
- Use a bottom-up dynamic programming approach to fill in the `dp` array for steps beyond the base cases.
- The loop starts from `i = 3` and iterates up to `n`. For each step `i`, the number of ways to reach it is the sum of the ways to reach the previous two steps (`dp[i - 1]` and `dp[i - 2]`).
- Return the result stored in `dp[n]`, which represents the number of distinct ways to climb to the top step.

2. Memoization and Tabulation:

- Problem Type: Use memoization (top-down) or tabulation (bottom-up) to optimize recursive solutions.

i. Memoization (Top-Down) Solution:

```
# Example Problem: “Fibonacci Number”
# The Fibonacci numbers are defined by the recurrence relation F(n) = F(n-1) + F

def fibonacci_memoization(n, memo={}):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1

    # Check if the result is already in the memoization table
    if n not in memo:
        # If not, calculate the result and store it in the memoization table
        memo[n] = fibonacci_memoization(n - 1, memo) + fibonacci_memoization(n -

    return memo[n]

# Example usage
n = 5
result = fibonacci_memoization(n)
print(f"Fibonacci number at position {n}: {result}")
```

- Function Definition: We define a function `fibonacci_memoization` that takes an integer `n` (the position in the Fibonacci sequence) and an optional memoization table `memo` (defaulted to an empty dictionary).
- Base Cases: We handle the base cases where `n` is 0 or 1 by directly returning 0 and 1, respectively.
- Memoization Check: We check if the result for the current `n` is already stored in the `memo` dictionary. If it is not present, we proceed to calculate it.
- Recursion: We use recursion to calculate the Fibonacci number at position `n` by summing the results of the two previous Fibonacci numbers (`n - 1` and `n - 2`).
- Memoization: We store the result in the `memo` dictionary to avoid redundant calculations and improve efficiency.
- Return: We return the Fibonacci number for the given position `n`.

ii. Tabulation (Bottom-Up) Solution:

```
def fibonacci_tabulation(n):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1

    # Initialize an array to store the Fibonacci numbers
    fib = [0] * (n + 1)

    # Base cases for 0 and 1
    fib[0] = 0
    fib[1] = 1

    # Fill in the array using a bottom-up approach
    for i in range(2, n + 1):
        fib[i] = fib[i - 1] + fib[i - 2]

    return fib[n]

# Example usage
n = 5
result = fibonacci_tabulation(n)
print(f"Fibonacci number at position {n}: {result}")
```

- Function Definition: We define a function `fibonacci_tabulation` that takes an integer `n` (the position in the Fibonacci sequence).
- Base Cases: We handle the base cases where `n` is 0 or 1 by directly returning 0 and 1, respectively.
- Array Initialization: We initialize an array `fib` of length `n + 1` to store the Fibonacci numbers.
- Base Case Values: We set the base case values directly in the array (`fib[0]` and `fib[1]`).
- Fill in the Array: We use a bottom-up approach to fill in the array by iterating from `i = 2` to `n`. At each step, we calculate the Fibonacci number at position `i` by summing the previous two Fibonacci numbers (`fib[i - 1]` and `fib[i - 2]`).
- Return: We return the Fibonacci number for the given position `n`.

These implementations demonstrate how both memoization and tabulation can be applied to efficiently solve the Fibonacci Number problem, avoiding redundant calculations and improving overall performance.

3. Subset and Subsequence Problems:

- Problem Type: Find subsets or subsequences with specific properties.

- DP Concept: Similar to the knapsack problem, this solution uses dynamic programming to determine if there exists a subset that sums up to a target sum.
- DP Steps: A 2D array (`dp`) is filled bottom-up by considering whether a subset with a given sum can be formed using different elements of the input set.

```
# Example Problem: “Subset Sum”
# Given a set of positive integers and a target sum, determine if there is any n

def subset_sum(nums, target_sum):
    # Create a 2D array to store the subset sum information
    n = len(nums)
    dp = [[False] * (target_sum + 1) for _ in range(n + 1)]

    # Base case: an empty subset can always achieve a sum of 0
    for i in range(n + 1):
        dp[i][0] = True

    # Fill in the array using a bottom-up approach
    for i in range(1, n + 1):
        for j in range(1, target_sum + 1):
            # If the current element is greater than the target sum, exclude it
            if nums[i - 1] > j:
                dp[i][j] = dp[i - 1][j]
            else:
                # Include the current element or exclude it, based on previous r
                dp[i][j] = dp[i - 1][j] or dp[i - 1][j - nums[i - 1]]

    # The final result is in the bottom-right corner of the matrix
    return dp[n][target_sum]

# Example usage
nums = [3, 34, 4, 12, 5, 2]
target_sum = 9
result = subset_sum(nums, target_sum)
print(f"There exists a subset with sum {target_sum}: {result}")
```

- Function Definition: We define a function `subset_sum` that takes a list of positive integers `nums` and a target sum `target_sum` .
- Array Initialization: We create a 2D array `dp` of size `(n + 1) x (target_sum + 1)` . The rows represent the elements in the `nums` list, and the columns represent the possible target sums.
- Base Case: We set the base case where an empty subset can always achieve a sum of 0. Therefore, for any row `i` , `dp[i][0]` is set to `True` .
- Dynamic Programming (Bottom-Up): We use a bottom-up dynamic programming approach to fill in the `dp` array. We iterate through each

element in `nums` (represented by `i`) and each possible target sum (represented by `j`).

- If the current element (`nums[i - 1]`) is greater than the target sum `j`, it cannot be included in the subset, so we exclude it by setting `dp[i][j] = dp[i - 1][j]`.
- Otherwise, we have the choice to include or exclude the current element. We set `dp[i][j]` based on the results of the previous row (`dp[i - 1][j]`) and the value at `j - nums[i - 1]` (representing the remaining sum if the current element is included).
- Result Retrieval: The final result is found in the bottom-right corner of the `dp` matrix and represents whether there exists a non-empty subset that sums up to the target sum.

4. Longest Increasing Subsequence (LIS):

- Problem Type: Find the length of the longest increasing subsequence in an array.
- DP Concept: The solution uses dynamic programming to find the length of the longest increasing subsequence in an array.
- DP Steps: A 1D array (`dp`) is filled bottom-up by considering the length of the longest increasing subsequence ending at each element of the array.

```
# Example Problem: “Longest Increasing Subsequence”
# Given an unsorted array of integers, find the length of the longest increasing
def length_of_lis(nums):
    if not nums:
        return 0

    n = len(nums)
    dp = [1] * n  # Initialize an array to store the lengths of increasing subse

    for i in range(1, n):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)

# Example usage
nums = [10, 9, 2, 5, 3, 7, 101, 18]
result = length_of_lis(nums)
print(f"Length of the longest increasing subsequence: {result}")
```

- Function Definition: We define a function `length_of_lis` that takes a list of integers `nums` as its input.

- Base Case Check: If the input list `nums` is empty, we return 0 because there is no increasing subsequence.
- Initialization: We get the length of the input list `nums` and initialize an array `dp` of length `n` to store the lengths of increasing subsequences. We initialize each element of `dp` to 1 because each element in the array is itself a valid increasing subsequence of length 1.
- Dynamic Programming: We use a nested loop to iterate over each pair of indices `(i, j)` in the array. For each pair, we check if the element at index `i` is greater than the element at index `j`. If it is, we update the length of the increasing subsequence ending at index `i` (`dp[i]`) by taking the maximum of its current length and the length of the increasing subsequence ending at index `j` plus 1.
- Result Retrieval: We return the maximum value in the `dp` array, which represents the length of the longest increasing subsequence.

5. Edit Distance:

- Problem Type: Calculate the minimum number of operations (insert, delete, replace) required to transform one string into another.
- DP Concept: The solution applies dynamic programming to calculate the minimum number of operations (insert, delete, replace) required to transform one string into another.
- DP Steps: A 2D array (`dp`) is filled bottom-up by considering the minimum operations required for transforming substrings of the two input strings.

```
# Example Problem: “Edit Distance”
# Given two words word1 and word2, find the minimum number of operations require

def min_distance(word1, word2):
    m, n = len(word1), len(word2)

    # Initialize a 2D array to store the minimum edit distances
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the base cases
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    # Fill in the array using a bottom-up approach
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # If the current characters are equal, no operation needed
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
```

```
                else:
                    # Choose the minimum of insert, delete, or replace operations
                    dp[i][j] = 1 + min(dp[i - 1][j],  # Delete
                                       dp[i][j - 1],  # Insert
                                       dp[i - 1][j - 1])  # Replace

        # The final result is in the bottom-right corner of the matrix
        return dp[m][n]

    # Example usage
    word1 = "horse"
    word2 = "ros"
    result = min_distance(word1, word2)
    print(f"Minimum number of operations: {result}")
```

- Function Definition: We define a function `min_distance` that takes two strings, `word1` and `word2`, as its input.
- Initialization: We get the lengths of the two words (`m` and `n`) and initialize a 2D array `dp` of size `(m + 1) x (n + 1)` to store the minimum edit distances. The rows represent the characters in `word1`, and the columns represent the characters in `word2`.
- Base Cases: We initialize the base cases for the first row and first column of the `dp` array. The values in the first row represent the minimum number of operations needed to convert an empty string to a prefix of `word1`, and the values in the first column represent the minimum number of operations needed to convert an empty string to a prefix of `word2`.
- Dynamic Programming (Bottom-Up): We use a nested loop to fill in the `dp` array. For each pair of indices `(i, j)`, we check if the current characters in `word1` and `word2` are equal. If they are equal, no operation is needed, and the value is copied from the diagonal (`dp[i - 1][j - 1]`). If they are not equal, we choose the minimum of the three possible operations (delete, insert, or replace).
- Result Retrieval: We return the value in the bottom-right corner of the `dp` array, which represents the minimum number of operations needed to convert `word1` to `word2`.

6. Knapsack Problems:

- Problem Type: Optimize the selection of items to maximize or minimize a value within a given capacity.
- DP Concept: The solution employs dynamic programming to optimize the selection of items to maximize the total value within a given capacity.

- DP Steps: A 2D array (`dp`) is filled in a bottom-up manner by considering the maximum value achievable with different combinations of items and capacities.

```
# Example Problem: “0/1 Knapsack”
# Given weights and values of n items, put these items in a knapsack of capacity

def knapsack(weights, values, capacity):
    n = len(weights)

    # Initialize a 2D array to store the maximum values for different capacities
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    # Fill in the array using a bottom-up approach
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            # If the current item's weight is more than the current capacity, sk
            if weights[i - 1] > w:
                dp[i][w] = dp[i - 1][w]
            else:
                # Choose the maximum value between including and excluding the c
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weigh

    return dp[n][capacity]

# Example usage
weights = [2, 1, 3]
values = [4, 2, 3]
capacity = 4
result = knapsack(weights, values, capacity)
print(f"Maximum value in the knapsack: {result}")
```

- Function Definition: We define a function `knapsack` that takes three lists as input: `weights` (weights of the items), `values` (values of the items), and `capacity` (the capacity of the knapsack).
- Initialization: We get the number of items (`n`) and initialize a 2D array `dp` of size `(n + 1) x (capacity + 1)` to store the maximum values for different capacities. The rows represent the items, and the columns represent the capacities.
- Dynamic Programming (Bottom-Up): We use nested loops to fill in the `dp` array. For each pair of indices `(i, w)` , we check if the current item's weight (`weights[i - 1]`) is more than the current capacity (`w`). If it is, we skip the item. Otherwise, we choose the maximum value between including and excluding the current item.
- If we exclude the current item, the value is taken from the cell above (`dp[i - 1][w]`).

- If we include the current item, the value is the sum of the current item's value and the value from the cell in the same row but with reduced capacity (`dp[i - 1][w - weights[i - 1]]`).
- Result Retrieval: We return the value in the bottom-right corner of the `dp` array, which represents the maximum value that can be achieved with the given capacity.

7. Matrix Chain Multiplication:

- Problem Type: Optimize the parenthesization of matrix multiplications to minimize the number of scalar multiplications.
- DP Concept: The solution uses a 2D array (`dp`) to store the minimum number of scalar multiplications for multiplying matrices in different subchains.
- DP Steps: The array is filled in a bottom-up manner by considering the minimum cost of multiplying matrices for smaller subchains and building up to the final solution.

```
# Example Problem: “Matrix Chain Multiplication”
# Given a sequence of matrices, find the most efficient way to multiply these ma

def matrix_chain_multiplication(dimensions):
    n = len(dimensions) - 1  # Number of matrices

    # Initialize a 2D array to store the minimum number of scalar multiplication
    dp = [[float('inf')] * n for _ in range(n)]

    # The cost of multiplying one matrix is always 0
    for i in range(n):
        dp[i][i] = 0

    # Fill in the array using a bottom-up approach
    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            for k in range(i, j):
                cost = dp[i][k] + dp[k + 1][j] + dimensions[i] * dimensions[k +
                dp[i][j] = min(dp[i][j], cost)

    return dp[0][n - 1]

# Example usage
matrix_dimensions = [30, 35, 15, 5, 10, 20, 25]
result = matrix_chain_multiplication(matrix_dimensions)
print(f"Minimum number of scalar multiplications: {result}")
```

- Function Definition: We define a function `matrix_chain_multiplication` that takes a list `dimensions` representing the dimensions of matrices in

the chain.

- Initialization: We get the number of matrices (`n`) and initialize a 2D array `dp` of size `n x n` to store the minimum number of scalar multiplications. Each entry `dp[i][j]` will represent the minimum cost of multiplying matrices from index `i` to `j` .
- Base Case: The cost of multiplying a single matrix is always 0, so we initialize the diagonal entries of the `dp` array to 0.
- Dynamic Programming (Bottom-Up): We use nested loops to fill in the `dp` array. The outer loop (`length`) iterates over the length of the matrix chain. The middle loop (`i`) iterates over the starting index of the chain, and the inner loop (`k`) iterates over the possible positions to split the chain. We calculate the cost of multiplying matrices from `i` to `k` and from `k + 1` to `j` and add the cost of multiplying the resulting matrices. We then update `dp[i][j]` with the minimum cost.
- Result Retrieval: We return the value in the top-right corner of the `dp` array, which represents the minimum number of scalar multiplications needed to multiply all matrices in the chain.

8. Longest Common Subsequence (LCS):

- Problem Type: Find the length of the longest common subsequence in two strings.
- DP Concept: The solution involves using a 2D array (`dp`) to store the lengths of common subsequences for different pairs of indices in the input strings.
- DP Steps: The array is filled in a bottom-up manner by considering the lengths of common subsequences for smaller substrings and building up to the final solution.

```
# Example Problem: “Longest Common Subsequence”
# Given two strings text1 and text2, return the length of their longest common s

def longest_common_subsequence(text1, text2):
    m, n = len(text1), len(text2)

    # Initialize a 2D array to store the lengths of common subsequences
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Fill in the array using a bottom-up approach
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
```

```
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]

# Example usage
text1 = "abcde"
text2 = "ace"
result = longest_common_subsequence(text1, text2)
print(f"Length of the longest common subsequence: {result}")
```

- Function Definition: We define a function `longest_common_subsequence` that takes two strings, `text1` and `text2`, as its input.
- Initialization: We get the lengths of the two strings (`m` and `n`) and initialize a 2D array `dp` of size `(m + 1) x (n + 1)` to store the lengths of common subsequences. The rows represent the characters in `text1`, and the columns represent the characters in `text2`.
- Dynamic Programming (Bottom-Up): We use nested loops to fill in the `dp` array. For each pair of indices `(i, j)`, we check if the current characters in `text1` and `text2` are equal. If they are, we extend the length of the common subsequence by 1 (`dp[i][j] = dp[i - 1][j - 1] + 1`). Otherwise, we take the maximum of the lengths obtained by excluding the current character in either string (`dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])`).
- Result Retrieval: We return the value in the bottom-right corner of the `dp` array, which represents the length of the longest common subsequence.

9. Palindrome Problems:

- Problem Type: Find the longest palindromic subsequence or the minimum number of insertions/deletions to make a string palindrome.
- DP Concept: Similar to LCS, the solution uses a 2D array (`dp`) to store the lengths of palindromic subsequences for different substrings.
- DP Steps: The array is filled bottom-up, considering the lengths of palindromic subsequences for smaller substrings and building up to the length of the longest palindromic subsequence.

```
# Problem: “Longest Palindromic Subsequence”
# Given a string s, find the length of the longest palindromic subsequence in s.

def longest_palindromic_subsequence(s):
    n = len(s)

    # Initialize a 2D array to store the lengths of palindromic subsequences
    dp = [[0] * n for _ in range(n)]
```

```
        # All substrings of length 1 are palindromes
        for i in range(n):
            dp[i][i] = 1

        # Fill in the array using a bottom-up approach
        for length in range(2, n + 1):
            for i in range(n - length + 1):
                j = i + length - 1
                if s[i] == s[j] and length == 2:
                    dp[i][j] = 2
                elif s[i] == s[j]:
                    dp[i][j] = dp[i + 1][j - 1] + 2
                else:
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

        return dp[0][n - 1]

    # Example usage
    s = "bbbab"
    result = longest_palindromic_subsequence(s)
    print(f"Length of the longest palindromic subsequence: {result}")
```

- Function Definition: We define a function `longest_palindromic_subsequence` that takes a string `s` as its input.
- Initialization: We get the length of the string (`n`) and initialize a 2D array `dp` of size `n x n` to store the lengths of palindromic subsequences. Each entry `dp[i][j]` will represent the length of the palindromic subsequence in the substring from index `i` to `j` .
- Base Case: All substrings of length 1 are palindromes, so we initialize the diagonal entries of the `dp` array to 1.
- Dynamic Programming (Bottom-Up): We use nested loops to fill in the `dp` array. The outer loop (`length`) iterates over the length of the substring. The middle loop (`i`) iterates over the starting index of the substring, and the inner loop (`j`) iterates over the possible ending index of the substring. We check if the characters at the current indices form a palindrome and update the length accordingly.
- Result Retrieval: We return the value in the top-right corner of the `dp` array, which

10. Dynamic Programming on Trees:

- Problem Type: Optimize tree-related problems using dynamic programming techniques.
- DP Concept: The solution employs dynamic programming to optimize the robbing of houses in a binary tree structure.

- DP Steps: A recursive approach is used to traverse the tree in a bottom-up manner, calculating the maximum stolen value with and without robbing each house. Results are stored and reused to avoid redundant computations.

```
# Problem: House Robber III
# The thief has found himself a new place for his thievery again. There is only

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def rob(root):
    def helper(node):
        if not node:
            return (0, 0)

        # Recursive calls to the left and right subtrees
        left_with, left_without = helper(node.left)
        right_with, right_without = helper(node.right)

        # Calculate the maximum value with and without robbing the current house
        with_current = node.val + left_without + right_without
        without_current = max(left_with, left_without) + max(right_with, right_w

        return (with_current, without_current)

    result_with, result_without = helper(root)
    return max(result_with, result_without)

# Example usage
# Construct a sample tree: [3,2,3,null,3,null,1]
root = TreeNode(3)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.right = TreeNode(3)
root.right.right = TreeNode(1)

result = rob(root)
print(f"Maximum stolen value: {result}")
```

- TreeNode Definition: We define a simple class `TreeNode` to represent nodes in the binary tree. Each node has a value (`val`) and references to its left and right children.
- rob Function Definition: We define a function `rob` that takes the root of a binary tree as input.
- Helper Function: We define a helper function `helper` that calculates the maximum stolen value for a subtree rooted at a given node. The function returns a tuple `(with_current, without_current)` representing the maximum value with and without robbing the current house.

- Dynamic Programming (Bottom-Up): The function uses recursion to traverse the tree in a bottom-up manner. For each node, it calculates the maximum value with and without robbing the current house and updates the results accordingly.

Leetcode | Leetcode Solution | Coding Interviews | Python | Dynamic Programming

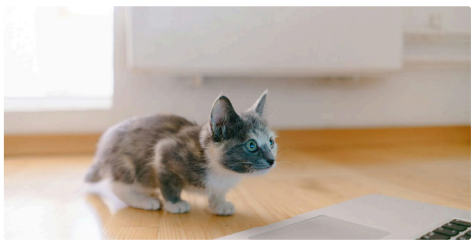
Written by btd

866 Followers

Learning & making lists

Follow

More from btd



btd

100 Deep Learning Technical Interview Questions and SHORT...

I. Basics of Deep Learning:

✦ · 14 min read · Nov 17, 2023

135

btd

12 Probability Distributions in Data Science

In the context of data science, distributions refer to the patterns by which data values ar...

✦ · 13 min read · Nov 11, 2023

103