# Minimum Elements for Sum

Let's break down the provided code and explain the logic step-by-step to ensure complete understanding. I'll include comments in the code and provide a detailed explanation of the algorithm.

## Detailed Explanation

### Problem Statement

Given an array of integers and a target sum `T`, the goal is to find the minimum number of elements from the array that sum up to `T`. If it's not possible to achieve the target sum, return `-1`.

## Algorithm

1. **Base Case:**
   - If the current index is `0`, check if the target `T` is divisible by the first element of the array:
     - If yes, return the quotient `T / arr[0]`.
     - If no, return a large value to indicate it's not possible (using `10^9` as infinity).
2. **Recursive Case:**
   - If the subproblem result is already computed (cached in `dp`), return it.
   - Compute the minimum elements required by:
     - **Not taking** the current element.
     - **Taking** the current element if it's less than or equal to `T`.
   - Store and return the minimum of the two computed values.
3. **Initialization and Result:**
   - Initialize the `dp` array to `-1` to indicate unsolved subproblems.
   - Calculate the result for the entire problem.
   - If the result is greater than or equal to `10^9`, return `-1`.

## Java Code with Detailed Comments

```java
import java.util.*;

class TUF {
    // Recursive function to find the minimum number of elements to achieve the target sum
    static int minimumElementsUtil(int[] arr, int ind, int T, int[][] dp) {
        // Base case: If the current index is 0
```

```
        if (ind == 0) {
            // If T is divisible by the first element of the array, return the quotient
            if (T % arr[0] == 0)
                return T / arr[0];
            else
                // If not, return a large value to indicate it's not possible
                return (int) Math.pow(10, 9);
        }

        // If the result for this subproblem has already been calculated, return it
        if (dp[ind][T] != -1)
            return dp[ind][T];

        // Calculate the minimum number of elements needed without taking the current
element
        int notTaken = 0 + minimumElementsUtil(arr, ind - 1, T, dp);

        // Initialize the minimum number of elements needed taking the current element
        int taken = (int) Math.pow(10, 9);

        // If the current element is less than or equal to T, calculate the minimum taking
it
        if (arr[ind] <= T)
            taken = 1 + minimumElementsUtil(arr, ind, T - arr[ind], dp);

        // Store the minimum result in the dp array and return it
        return dp[ind][T] = Math.min(notTaken, taken);
    }

    // Function to find the minimum number of elements to achieve the target sum
    static int minimumElements(int[] arr, int T) {
        int n = arr.length;

        // Create a 2D array to store results of subproblems
        int[][] dp = new int[n][T + 1];

        // Initialize the dp array with -1 to indicate that subproblems are not solved yet
        for (int[] row : dp)
            Arrays.fill(row, -1);

        // Calculate the minimum number of elements to achieve the target sum
        int ans = minimumElementsUtil(arr, n - 1, T, dp);

        // If it's not possible to achieve the target sum, return -1
        if (ans >= (int) Math.pow(10, 9))
            return -1;
        return ans;
    }

    public static void main(String[] args) {
        int arr[] = { 1, 2, 3 };
        int T = 7;

        // Call the minimumElements function and print the result
        System.out.println("The minimum number of coins required to form the target sum is "
+ minimumElements(arr, T));
    }
}
```

## Dry Run Example

Let's dry run the provided example with `arr = {1, 2, 3}` and `T = 7`:

1. Start with the last index (2, element `3`) and target `T = 7`.
2. Check two cases: taking or not taking `3`.

- Not taking `3`: Solve for `index = 1` and `T = 7`.
- Taking `3`: Solve for `index = 2` and `T = 4` (`7 - 3`).

3. Continue recursively until base cases are reached and combine results.
4. The recursive calls and memoization will ensure we find the optimal solution.

This approach ensures that we check all possible combinations and use memoization to avoid redundant calculations, leading to an efficient solution.

Let's dive into the dynamic programming (DP) approach used in the provided code. I'll explain each step in detail to ensure you fully understand the logic and how it is implemented in the code.

## Problem Statement

Given an array of integers and a target sum `T`, find the minimum number of elements from the array that sum up to `T`. If it's not possible to achieve the target sum, return `-1`.

## Dynamic Programming Approach

### Explanation

1. **DP Table Initialization:**
   - Create a 2D array `dp` where `dp[i][j]` represents the minimum number of elements needed to achieve the sum `j` using the first `i` elements of the array.
   - Initialize the first row of the DP table based on the first element of the array.
2. **Filling the DP Table:**
   - Iterate through each element of the array.
   - For each element, iterate through all possible target sums from `0` to `T`.
   - For each target sum, calculate the minimum number of elements needed:
     - **Not Taking** the current element.
     - **Taking** the current element if it's less than or equal to the target sum.
3. **Result Extraction:**
   - The result is found in `dp[n-1][T]` where `n` is the number of elements in the array.
   - If the result is a large value (indicating it's not possible to achieve the target sum), return `-1`.

## Detailed Code with Comments

```java
import java.util.*;
```

```java
class TUF {
    // Function to find the minimum number of elements to achieve the target sum
    static int minimumElements(int[] arr, int T) {
        int n = arr.length;

        // Create a 2D array to store results of subproblems
        int[][] dp = new int[n][T + 1];

        // Initialize the dp array for the first element of the array
        for (int i = 0; i <= T; i++) {
            if (i % arr[0] == 0) {
                // If i is divisible by arr[0], fill with the quotient
                dp[0][i] = i / arr[0];
            } else {
                // Otherwise, fill with a large value indicating it's not possible
                dp[0][i] = (int) Math.pow(10, 9);
            }
        }

        // Fill the dp array using dynamic programming
        for (int ind = 1; ind < n; ind++) {
            for (int target = 0; target <= T; target++) {
                // Case 1: Not taking the current element
                int notTake = 0 + dp[ind - 1][target];

                // Case 2: Taking the current element (initialize with a large value)
                int take = (int) Math.pow(10, 9);

                // If the current element is less than or equal to the target, calculate
'take'
                if (arr[ind] <= target) {
                    take = 1 + dp[ind][target - arr[ind]];
                }

                // Store the minimum result in the dp array
                dp[ind][target] = Math.min(notTake, take);
            }
        }

        // Get the minimum number of elements needed for the target sum
        int ans = dp[n - 1][T];

        // If it's not possible to achieve the target sum, return -1
        if (ans >= (int) Math.pow(10, 9)) {
            return -1;
        }
        return ans;
    }

    public static void main(String[] args) {
        int arr[] = { 1, 2, 3 };
        int T = 7;

        // Call the minimumElements function and print the result
        System.out.println("The minimum number of coins required to form the target sum is "
+ minimumElements(arr, T));
    }
}
```

## Detailed Step-by-Step Explanation

1. **Initialization:**
    - The first row of the DP table is initialized based on the first element of the array.

- For each value of `T` (from `0` to `T`), if `T` is divisible by the first element `arr[0]`, set `dp[0][T] = T / arr[0]`. Otherwise, set it to a large value (`10^9`) indicating that it's not possible.

2. **Filling the DP Table:**

   - Iterate over the remaining elements of the array (`ind` from `1` to `n-1`).
   - For each element, iterate over all possible target values (`target` from `0` to `T`).
   - For each `target`, calculate two scenarios:
     - **Not Taking** the current element: `notTake = dp[ind-1][target]`.
     - **Taking** the current element: `take = 1 + dp[ind][target - arr[ind]]` (only if `arr[ind] <= target`).
   - Store the minimum of the two values in `dp[ind][target]`.

3. **Result Extraction:**

   - The result for the entire problem is stored in `dp[n-1][T]`.
   - If `dp[n-1][T]` is greater than or equal to `10^9`, return `-1` indicating it's not possible to achieve the target sum. Otherwise, return the result.

## Dry Run Example

For `arr = {1, 2, 3}` and `T = 7`:

1. **Initialization:**

   - `dp[0][0] = 0`, `dp[0][1] = 1`, `dp[0][2] = 2`, ..., `dp[0][7] = 7`.

2. **Filling DP Table:**

   - For `ind = 1` (element `2`):
     - Calculate `dp[1][0]`, `dp[1][1]`, ..., `dp[1][7]` based on the values of `dp[0][0]` to `dp[0][7]` and the element `2`.
   - For `ind = 2` (element `3`):
     - Calculate `dp[2][0]`, `dp[2][1]`, ..., `dp[2][7]` based on the values of `dp[1][0]` to `dp[1][7]` and the element `3`.

3. **Result Extraction:**

   - The final answer is found in `dp[2][7]`.

This approach efficiently finds the minimum number of elements needed to achieve the target sum using dynamic programming by breaking the problem into smaller subproblems and combining their solutions.

The initialization of the first row of the DP table explicitly is crucial because it sets the foundation for the rest of the table. In dynamic programming, the base case provides the starting point, and all other values are built upon these initial values. Here's a detailed explanation of why and how we initialize the first row explicitly:

## Why Initialize the First Row Explicitly?

1. **Base Case Setup:**
   - The first row corresponds to using only the first element of the array to form various target sums. This forms the base case of our dynamic programming solution.
   - Properly setting this base case ensures that subsequent rows (which build upon this base case) are calculated correctly.

2. **Single Element Analysis:**
   - When we are only allowed to use the first element (`arr[0]`), the only way to form a target sum `T` is by using multiples of `arr[0]`.
   - If `T` is divisible by `arr[0]`, the number of elements needed is `T / arr[0]`.
   - If `T` is not divisible by `arr[0]`, it's impossible to form that sum with just `arr[0]`, so we use a large value (`10^9`) to indicate this impossibility.

## How to Initialize the First Row?

1. **Iterate Through All Possible Target Sums:**
   - For each possible target sum from `0` to `T`, check if it can be formed using the first element (`arr[0]`).
2. **Fill Values Based on Divisibility:**
   - If `target % arr[0] == 0`, set `dp[0][target] = target / arr[0]`.
   - Otherwise, set `dp[0][target] = 10^9` to indicate it's not possible to form that target with just the first element.

## Example Code with Explanation

Here's the code with detailed comments explaining the initialization and the rest of the DP table filling:

java

```java
import java.util.*;

class TUF {
    // Function to find the minimum number of elements to achieve the target sum
    static int minimumElements(int[] arr, int T) {
        int n = arr.length;

        // Create a 2D array to store results of subproblems
        int[][] dp = new int[n][T + 1];

        // Initialize the dp array for the first element of the array
        for (int i = 0; i <= T; i++) {
            if (i % arr[0] == 0) {
                // If i is divisible by arr[0], fill with the quotient
```

```
                dp[0][i] = i / arr[0];
            } else {
                // Otherwise, fill with a large value indicating it's not possible
                dp[0][i] = (int) Math.pow(10, 9);
            }
        }

        // Fill the dp array using dynamic programming for the rest of the elements
        for (int ind = 1; ind < n; ind++) {
            for (int target = 0; target <= T; target++) {
                // Case 1: Not taking the current element
                int notTake = dp[ind - 1][target];

                // Case 2: Taking the current element (initialize with a large value)
                int take = (int) Math.pow(10, 9);

                // If the current element is less than or equal to the target, calculate
'take'
                if (arr[ind] <= target) {
                    take = 1 + dp[ind][target - arr[ind]];
                }

                // Store the minimum result in the dp array
                dp[ind][target] = Math.min(notTake, take);
            }
        }

        // Get the minimum number of elements needed for the target sum
        int ans = dp[n - 1][T];

        // If it's not possible to achieve the target sum, return -1
        if (ans >= (int) Math.pow(10, 9)) {
            return -1;
        }
        return ans;
    }

    public static void main(String[] args) {
        int arr[] = { 1, 2, 3 };
        int T = 7;

        // Call the minimumElements function and print the result
        System.out.println("The minimum number of coins required to form the target sum is "
+ minimumElements(arr, T));
    }
}
```

## Detailed Step-by-Step Initialization

1. **Initialization:**
   - For each possible target sum `i` from `0` to `T`:
     - If `i` is divisible by `arr[0]`, set `dp[0][i] = i / arr[0]`.
     - Otherwise, set `dp[0][i] = 10^9`.

2. **Example Initialization for `arr = {1, 2, 3}` and `T = 7`:**
   - `dp[0][0] = 0` (0 is divisible by 1, so 0 elements are needed)
   - `dp[0][1] = 1` (1 is divisible by 1, so 1 element is needed)
   - `dp[0][2] = 2` (2 is divisible by 1, so 2 elements are needed)
   - ...
   - `dp[0][7] = 7` (7 is divisible by 1, so 7 elements are needed)

By correctly setting up the base case, we ensure that all subsequent calculations in the DP table are based on accurate initial values, leading to the correct solution for the overall problem.

To determine which parts of the DP table (first row, first column, or specific cells) need to be explicitly initialized, you need to consider the nature of the problem and the base cases for the recursive relation. Here are some general guidelines and examples to help you understand how to handle different types of DP problems:

## General Guidelines

1. **Understand the Recurrence Relation:**
    - Identify the base cases in your recurrence relation. These are the cases where the solution can be determined without recursion or iteration.
2. **Identify the Dimensions of the DP Table:**
    - The dimensions of your DP table depend on the parameters of your problem. Typically, this could involve indices of items, target sums, lengths of strings, etc.
3. **Initialize Base Cases:**
    - Base cases are typically the minimal subproblems that can be solved directly. These will be the starting points for filling up the DP table.
4. **Fill the DP Table Based on Dependencies:**
    - Each cell in the DP table is filled based on the values of previously filled cells. Ensure that you follow the order which respects these dependencies.

## Specific Scenarios and Examples

### 1. Coin Change Problem (Minimum Number of Coins)

**Problem:**
Given a set of coins and a target sum, find the minimum number of coins needed to achieve the target sum.

**DP Table:**

- `dp[i][j]` represents the minimum number of coins needed to make sum `j` using the first `i` coins.

**Base Case Initialization:**

- First row (`dp[0][j]`): When using only the first type of coin.
- `dp[0][0] = 0` because zero coins are needed to make sum zero.
- For other sums `j`, if `j % arr[0] == 0`, `dp[0][j] = j / arr[0]`, else `dp[0][j] = ∞` (indicating it's not possible).

```java
for (int j = 0; j <= T; j++) {
    if (j % arr[0] == 0) {
        dp[0][j] = j / arr[0];
    } else {
        dp[0][j] = (int) Math.pow(10, 9);
    }
}
```

### 2. Knapsack Problem

**Problem:**
Given weights and values of items, determine the maximum value that can be put in a knapsack of a given capacity.

**DP Table:**

- `dp[i][w]` represents the maximum value that can be obtained with the first `i` items and capacity `w`.

**Base Case Initialization:**

- First row (`dp[0][w]`): When using only the first item.
- First column (`dp[i][0]`): Zero capacity knapsack, hence value `0`.

```java
// Initialize first row for only the first item
for (int w = 0; w <= W; w++) {
    if (weights[0] <= w) {
        dp[0][w] = values[0];
    } else {
        dp[0][w] = 0;
    }
}

// Initialize first column for zero capacity knapsack
for (int i = 0; i < n; i++) {
    dp[i][0] = 0;
}
```

### 3. Longest Common Subsequence

**Problem:**
Given two strings, find the length of their longest common subsequence.

**DP Table:**

- `dp[i][j]` represents the length of the LCS of the first `i` characters of string `A` and the first `j` characters of string `B`.

**Base Case Initialization:**

- First row and first column initialized to `0`, as an LCS involving an empty string is `0`.

```java
for (int i = 0; i <= m; i++) {
    dp[i][0] = 0;
}
for (int j = 0; j <= n; j++) {
    dp[0][j] = 0;
}
```

## Example: Longest Common Subsequence (LCS)

Here's a complete code example for finding the LCS length, initializing the first row and first column explicitly:

```java
import java.util.*;

public class LCS {
    // Function to find the length of the longest common subsequence
    static int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length();
        int n = text2.length();

        // Create a 2D array to store lengths of LCS subproblems
        int[][] dp = new int[m + 1][n + 1];

        // Initialize the first row and first column with 0
        for (int i = 0; i <= m; i++) {
            dp[i][0] = 0;
        }
        for (int j = 0; j <= n; j++) {
            dp[0][j] = 0;
        }

        // Fill the dp array using dynamic programming
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        // Return the length of LCS for the entire strings
        return dp[m][n];
    }
```

```
        public static void main(String[] args) {
            String text1 = "abcde";
            String text2 = "ace";
            System.out.println("The length of the longest common subsequence is " +
    longestCommonSubsequence(text1, text2));
        }
    }
```

## Conclusion

To determine which cells to initialize explicitly in a DP table:

- Analyze the problem to identify the base cases.
- Set the initial values for these base cases directly in the DP table.
- Ensure that these initial values correctly represent the simplest subproblems, from which all other values in the table will be derived.
- Follow the recurrence relation to fill in the rest of the table, ensuring dependencies are respected.

By thoroughly understanding the problem and the recurrence relation, you can identify the appropriate cells to initialize explicitly in any DP problem.

Certainly! Let's break down the key difference between problems where elements can be used multiple times (unbounded problems) and those where each element can be used only once (bounded problems), using examples for clarity.

## Unbounded Problems (Elements Can Be Used Multiple Times)

In these problems, you can use each element (coin/item) as many times as you need. This means that if you decide to include an element, you do not move to the next element; you stay on the current element and reduce the target.

**Example: Coin Change Problem (Unbounded Knapsack)**

**Problem:**
Given a set of coin denominations, find the minimum number of coins needed to make a target amount.

**Recursive Relation:**

- If we do not take the current coin, we move to the next coin.
- If we take the current coin, we stay on the same coin and reduce the target.

```
java

  int notTaken = 0 + minimumElementsUtil(arr, ind - 1, T, dp);

  int taken = (int) Math.pow(10, 9);
```

```
    if (arr[ind] <= T) {
        taken = 1 + minimumElementsUtil(arr, ind, T - arr[ind], dp);
    }

    dp[ind][T] = Math.min(notTaken, taken);
```

**Explanation:**

- **notTaken:** We do not take the current coin (`arr[ind]`), so we move to the next coin by decrementing the index (`ind - 1`) and keep the target `T` unchanged.
- **taken:** We take the current coin (`arr[ind]`). Since we can use the same coin again, we do not move to the next coin (`ind` remains the same), and we reduce the target by the value of the current coin (`T - arr[ind]`).

## Bounded Problems (Each Element Can Be Used Only Once)

In these problems, you can use each element (coin/item) only once. This means that if you decide to include an element, you move to the next element and reduce the target.

**Example: 0/1 Knapsack Problem**

**Problem:**
Given weights and values of items, determine the maximum value that can be put in a knapsack of a given capacity.

**Recursive Relation:**

- If we do not take the current item, we move to the next item.
- If we take the current item, we move to the next item and reduce the capacity of the knapsack.

java

```
    int notTaken = 0 + knapsackUtil(weights, values, ind - 1, W, dp);

    int taken = (int) Math.pow(10, 9);
    if (weights[ind] <= W) {
        taken = values[ind] + knapsackUtil(weights, values, ind - 1, W - weights[ind], dp);
    }

    dp[ind][W] = Math.max(notTaken, taken);
```

**Explanation:**

- **notTaken:** We do not take the current item (`weights[ind]`), so we move to the next item by decrementing the index (`ind - 1`) and keep the capacity `W` unchanged.
- **taken:** We take the current item (`weights[ind]`). Since each item can only be used once, we move to the next item (`ind - 1`) and reduce the capacity of the knapsack by the weight of the current item (`W - weights[ind]`).

## Key Differences and Clear Explanation

### Unbounded (Coin Change / Unbounded Knapsack)

- **Stay on the same index if taking the element:** When an element is taken, it can be used again, so we stay on the same index and reduce the target.
- **Recursive call:**

```java
taken = 1 + minimumElementsUtil(arr, ind, T - arr[ind], dp);
```

Here, `ind` remains the same, meaning we can use the same element again.

### Bounded (0/1 Knapsack)

- **Move to the next index if taking the element:** When an element is taken, it cannot be used again, so we move to the next index and reduce the target/capacity.
- **Recursive call:**

```java
taken = values[ind] + knapsackUtil(weights, values, ind - 1, W - weights[ind], dp);
```

Here, `ind` is decremented, meaning we move to the next element.

## Visual Example

### Coin Change Example (Unbounded)

- Coins: {1, 2, 3}
- Target: 4

```scss
minimumElementsUtil(arr, 2, 4, dp)
= min(
    notTaken: minimumElementsUtil(arr, 1, 4, dp),
    taken: 1 + minimumElementsUtil(arr, 2, 1, dp)
)
```

Here, when we take the coin `3`, we stay on the same index `2` and reduce the target to `1`.

### 0/1 Knapsack Example (Bounded)

- Weights: {1, 2, 3}

- Values: {10, 20, 30}
- Capacity: 4

```scss
knapsackUtil(weights, values, 2, 4, dp)
= max(
    notTaken: knapsackUtil(weights, values, 1, 4, dp),
    taken: values[2] + knapsackUtil(weights, values, 1, 1, dp)
)
```

Here, when we take the item with weight `3`, we move to the next index `1` and reduce the capacity to `1`.

## Conclusion

- In **unbounded problems**, if you take an element, you stay on the same index and reduce the target.
- In **bounded problems**, if you take an element, you move to the next index and reduce the target/capacity.
  Understanding these differences is crucial for correctly implementing dynamic programming solutions for various types of problems.

Exactly! In the context of the coin change problem (or any similar unbounded problem where you can use elements multiple times), the difference between the `notTaken` and `taken` calculations is about whether we count the current coin or not. Let's break it down:

## Understanding `notTaken` and `taken`:

1. **notTaken**:
   - When we do **not** take the current coin, we do not count it towards our total. This means we just move to the next index (the previous coin in our array), and we do not decrement the target sum `T`.
   - Code:

     ```java
     int notTaken = 0 + minimumElementsUtil(arr, ind - 1, T, dp);
     ```

   - Here, `0 +` signifies that we are not counting this coin.
2. **taken**:

- When we **do** take the current coin, we count it towards our total. This means we stay on the current index (since we can reuse this coin) and decrement the target sum `T` by the value of the coin.
- Code:

```java
int taken = (int) Math.pow(10, 9);
if (arr[ind] <= T) {
    taken = 1 + minimumElementsUtil(arr, ind, T - arr[ind], dp);
}
```

- Here, `1 +` signifies that we are counting this coin.

## Visual Explanation:

### Unbounded Coin Change Problem

**Problem:**
Given an array of coin denominations and a target sum, find the minimum number of coins required to make the target sum.

Let's break down the decision process:

- **Coins:** {1, 2, 3}
- **Target:** 4

**Recursive Breakdown:**

1. **Base Case:** When the target sum `T` is `0`, we need `0` coins.
2. **Recurrence:**
   - If we do not take the current coin (`arr[ind]`), we simply move to the next coin: `ind - 1`.
   - If we take the current coin (`arr[ind]`), we stay on the same coin and reduce the target by the coin's value.

**Code:**

```java
static int minimumElementsUtil(int[] arr, int ind, int T, int[][] dp) {
    // Base case: If the current index is 0
    if (ind == 0) {
        // If T is divisible by the first element of the array, return the quotient
        if (T % arr[0] == 0)
            return T / arr[0];
        else
            // If not, return a large value to indicate it's not possible
            return (int) Math.pow(10, 9);
    }

    // If the result for this subproblem has already been calculated, return it
    if (dp[ind][T] != -1)
```

```
            return dp[ind][T];

        // Calculate the minimum number of elements needed without taking the current element
        int notTaken = 0 + minimumElementsUtil(arr, ind - 1, T, dp);

        // Initialize the minimum number of elements needed taking the current element
        int taken = (int) Math.pow(10, 9);

        // If the current element is less than or equal to T, calculate the minimum taking it
        if (arr[ind] <= T)
            taken = 1 + minimumElementsUtil(arr, ind, T - arr[ind], dp);

        // Store the minimum result in the dp array and return it
        return dp[ind][T] = Math.min(notTaken, taken);
    }
```

### Dry Run Example:

For `arr = {1, 2, 3}` and `T = 4`:

1. **Current Coin:** `3`, **Target:** `4`
    - **notTaken:** Move to next coin: `minimumElementsUtil(arr, 1, 4, dp)`
    - **taken:** Use current coin: `1 + minimumElementsUtil(arr, 2, 1, dp)`
2. **Recursive Calls:**
    - **notTaken:** `minimumElementsUtil(arr, 1, 4, dp)`
        - **Current Coin:** `2`, **Target:** `4`
            - **notTaken:** Move to next coin: `minimumElementsUtil(arr, 0, 4, dp)`
            - **taken:** Use current coin: `1 + minimumElementsUtil(arr, 1, 2, dp)`
    - **taken:** `1 + minimumElementsUtil(arr, 2, 1, dp)`
        - **Current Coin:** `3`, **Target:** `1`
            - Since `3 > 1`, we cannot take this coin.

### Summary:

- **notTaken:** Represents the scenario where the current coin is not counted. We move to the next coin (decrement the index) and keep the target the same.
- **taken:** Represents the scenario where the current coin is counted. We stay on the same coin (index remains the same) and reduce the target by the coin's value.

By understanding these scenarios, you can correctly implement the recursive logic for problems where elements can be used multiple times versus problems where elements can be used only once.

Sure! Here is the problem statement for the **Coin Change 2** problem:

### Coin Change 2

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Write a function to compute the number of combinations that make up that amount. If that amount of money cannot be made up by any combination of the coins, return `0`.

You may assume that you have an infinite number of each kind of coin.

**Example 1:**

```vbnet
Input: amount = 5, coins = [1, 2, 5]
Output: 4
Explanation: there are four ways to make up the amount:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

**Example 2:**

```vbnet
Input: amount = 3, coins = [2]
Output: 0
Explanation: the amount of 3 cannot be made up just with coins of 2.
```

**Example 3:**

```makefile
Input: amount = 10, coins = [10]
Output: 1
```

**Constraints:**

- `1 <= coins.length <= 300`
- `1 <= coins[i] <= 5000`
- All the values of `coins` are unique.
- `0 <= amount <= 5000`

Sure! Here is the `Coin Change 2` problem solved using a memoized (top-down) dynamic programming approach, with detailed comments explaining each step:

java

```java
import java.util.*;

class TUF {
    /**
     * Recursive function to count the ways to make change.
     *
     * @param arr The array of coin denominations.
     * @param ind The current index in the array.
     * @param T The remaining target amount.
     * @param dp The memoization table to store subproblem results.
     * @return The number of ways to make change for the target amount using coins up to the current index.
     */
    static long countWaysToMakeChangeUtil(int[] arr, int ind, int T, long[][] dp) {
        // Base case: If the current index is 0
        if (ind == 0) {
            // If T is divisible by the first element of the array, there is one way to make the change (using this coin repeatedly)
            if (T % arr[0] == 0)
                return 1;
            else
                // Otherwise, it is not possible to make the change
                return 0;
        }

        // If the result for this subproblem has already been calculated, return it
        if (dp[ind][T] != -1)
            return dp[ind][T];

        // Calculate the number of ways without taking the current element
        long notTaken = countWaysToMakeChangeUtil(arr, ind - 1, T, dp);

        // Initialize the number of ways taking the current element as 0
        long taken = 0;

        // If the current element is less than or equal to T, calculate the number of ways by taking it
        if (arr[ind] <= T)
            taken = countWaysToMakeChangeUtil(arr, ind, T - arr[ind], dp);

        // Store the result in the dp array and return it
        return dp[ind][T] = notTaken + taken;
    }

    /**
     * Function to count the ways to make change.
     *
     * @param arr The array of coin denominations.
     * @param n The number of coin denominations.
     * @param T The target amount.
     * @return The total number of ways to make change for the target amount.
     */
    static long countWaysToMakeChange(int[] arr, int n, int T) {
        // Create a 2D array to store results of subproblems
        long dp[][] = new long[n][T + 1];

        // Initialize the dp array with -1 to indicate that subproblems are not solved yet
        for (long[] row : dp)
            Arrays.fill(row, -1);

        // Call the countWaysToMakeChangeUtil function to calculate the number of ways
        return countWaysToMakeChangeUtil(arr, n - 1, T, dp);
```

```
    }

    public static void main(String[] args) {
        int arr[] = { 1, 2, 3 };
        int target = 4;
        int n = arr.length;

        // Call the countWaysToMakeChange function and print the result
        System.out.println("The total number of ways is " + countWaysToMakeChange(arr, n,
target));
    }
}
```

## Explanation:

1. **Base Case:**
   - If `ind == 0`, it means we are considering only the first coin (`arr[0]`).
   - If `T` is divisible by `arr[0]`, there is exactly one way to make the amount `T` using this coin repeatedly.
   - Otherwise, it is not possible to make the amount `T` using just `arr[0]`.
2. **Memoization:**
   - `dp[ind][T]` is used to store the result of subproblems to avoid recomputation.
   - If `dp[ind][T]` is not `-1`, it means we have already computed the result for this subproblem.
3. **Recursive Calls:**
   - **notTaken:** The number of ways to make the amount `T` without using the current coin (`arr[ind]`). We move to the previous coin (`ind - 1`).
   - **taken:** The number of ways to make the amount `T` by including the current coin (`arr[ind]`). We stay on the same coin (`ind`) but reduce the target amount by the coin's value (`T - arr[ind]`).
4. **Combination of Results:**
   - The total number of ways to make the amount `T` is the sum of the ways calculated by not taking the current coin and by taking the current coin.
   - `dp[ind][T] = notTaken + taken` stores this result in the memoization table.
5. **Initialization and Main Function:**
   - The `countWaysToMakeChange` function initializes the memoization table and calls the recursive utility function.
   - The `main` function tests the implementation with a sample input and prints the result.

Yes, that's correct! When we are dealing with problems where we need to **count the number of ways**, **combinations**, or **subsets** that satisfy a condition, our approach usually follows a specific pattern. The main idea is to combine the results from both the **"take"** and **"not take"** decisions.

Here's a structured explanation:

## Pattern for Counting Combinations or Subsets

1. **Base Cases**:
   - **Success Base Case**: Return `1` when the exact target is achieved (i.e., target becomes 0).
   - **Failure Base Case**: Return `0` when it's impossible to achieve the target (i.e., target is less than 0 or there are no items left to process).

2. **Recursive Case**:
   - **notTaken**: Count the ways without including the current item.
   - **taken**: Count the ways including the current item.
   - **Combine Results**: Sum the results of `notTaken` and `taken`.

3. **Memoization**:
   - Use a 2D array to store results of subproblems to avoid recomputation.

## Example Problems and Code Snippets

### Example 1: Coin Change 2 (Count Combinations)

**Problem Statement**: Given an integer array `coins` and an integer `amount`, compute the number of combinations that make up the amount. You have an infinite number of each coin.

**Code**:

java

```java
import java.util.*;

class CoinChange2 {
    static long countWaysToMakeChangeUtil(int[] coins, int index, int amount, long[][] dp) {
        // Base case: If the amount is 0, there is 1 way to make change (using no coins)
        if (amount == 0) return 1;

        // Base case: If the amount is less than 0 or no coins left to use, there is no way
to make change
        if (index < 0 || amount < 0) return 0;

        // If the result is already computed, return it
        if (dp[index][amount] != -1) return dp[index][amount];

        // Recursive calls
        long notTaken = countWaysToMakeChangeUtil(coins, index - 1, amount, dp);
        long taken = 0;
        if (coins[index] <= amount)
```

```
            taken = countWaysToMakeChangeUtil(coins, index, amount - coins[index], dp);

        // Combine results and store in dp array
        return dp[index][amount] = notTaken + taken;
    }

    static long countWaysToMakeChange(int[] coins, int amount) {
        int n = coins.length;
        long[][] dp = new long[n][amount + 1];

        for (long[] row : dp)
            Arrays.fill(row, -1);

        return countWaysToMakeChangeUtil(coins, n - 1, amount, dp);
    }

    public static void main(String[] args) {
        int[] coins = {1, 2, 3};
        int amount = 4;
        System.out.println("Total number of ways: " + countWaysToMakeChange(coins, amount));
    }
}
```

## Detailed Explanation:

1. **Base Cases**:
   - If `amount == 0`, it means we have found a valid combination of coins that add up to the target, so return `1`.
   - If `index < 0` (no coins left to consider) or `amount < 0` (target can't be achieved), return `0`.
2. **Recursive Case**:
   - **notTaken**: Don't take the current coin and move to the previous coin (`index - 1`).
   - **taken**: Take the current coin and stay on the same coin (`index`) while reducing the `amount` by the coin's value.
3. **Memoization**:
   - Use a 2D array `dp` where `dp[index][amount]` stores the number of ways to make the `amount` using coins from `0` to `index`.
4. **Combining Results**:
   - Sum the results of `notTaken` and `taken` because we are interested in counting all possible ways to achieve the target.

**Example 2: Subset Sum (Count Subsets)**

**Problem Statement**: Given an array of integers and a target sum, find the number of subsets that sum up to the target.

**Code**:

java

```java
import java.util.*;

class SubsetSum {
    static int countSubsetsUtil(int[] arr, int index, int sum, int[][] dp) {
        // Base case: If the sum is 0, there is 1 subset (the empty subset)
        if (sum == 0) return 1;

        // Base case: If no elements left or sum becomes negative, there is no subset
        if (index < 0 || sum < 0) return 0;

        // If the result is already computed, return it
        if (dp[index][sum] != -1) return dp[index][sum];

        // Recursive calls
        int notTaken = countSubsetsUtil(arr, index - 1, sum, dp);
        int taken = 0;
        if (arr[index] <= sum)
            taken = countSubsetsUtil(arr, index - 1, sum - arr[index], dp);

        // Combine results and store in dp array
        return dp[index][sum] = notTaken + taken;
    }

    static int countSubsets(int[] arr, int sum) {
        int n = arr.length;
        int[][] dp = new int[n][sum + 1];

        for (int[] row : dp)
            Arrays.fill(row, -1);

        return countSubsetsUtil(arr, n - 1, sum, dp);
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int sum = 5;
        System.out.println("Total number of subsets: " + countSubsets(arr, sum));
    }
}
```

## Detailed Explanation:

1. **Base Cases**:
   - If `sum == 0`, there is 1 valid subset (the empty subset), so return `1`.
   - If `index < 0` (no elements left) or `sum < 0` (target can't be achieved), return `0`.

2. **Recursive Case**:
   - **notTaken**: Don't include the current element and move to the previous element (`index - 1`).
   - **taken**: Include the current element and move to the previous element (`index - 1`) while reducing the `sum` by the element's value.

3. **Memoization**:
   - Use a 2D array `dp` where `dp[index][sum]` stores the number of subsets that sum up to `sum` using elements from `0` to `index`.

4. **Combining Results**:
    - Sum the results of `notTaken` and `taken` because we are counting all possible subsets that achieve the target.

## Summary:

In problems where we count combinations or subsets, our focus is on summing the results of taking or not taking elements, rather than minimizing or maximizing values. This pattern helps in solving a wide range of counting problems efficiently using dynamic programming with memoization.

Certainly! Here is the code with detailed comments explaining each step:

java

```java
import java.util.*;

class TUF {
    // Function to count the ways to make change
    static long countWaysToMakeChange(int[] arr, int n, int T) {
        // Create a 2D array to store results of subproblems
        long dp[][] = new long[n][T + 1];

        // Initialize base condition for the first element of the array
        for (int i = 0; i <= T; i++) {
            if (i % arr[0] == 0)
                dp[0][i] = 1; // If i is divisible by arr[0], there's one way to make change
using arr[0]
            // Else condition is automatically fulfilled, as dp array is initialized to zero
        }

        // Fill the dp array using dynamic programming
        for (int ind = 1; ind < n; ind++) {
            for (int target = 0; target <= T; target++) {
                // Count ways without taking the current element
                long notTaken = dp[ind - 1][target];

                // Count ways including the current element
                long taken = 0;
                if (arr[ind] <= target)
                    taken = dp[ind][target - arr[ind]];

                // Combine the ways of taking and not taking the current element
                dp[ind][target] = notTaken + taken;
            }
        }

        // Return the total number of ways to make the target sum
        return dp[n - 1][T];
    }

    public static void main(String args[]) {
        int arr[] = { 1, 2, 3 };
        int target = 4;
        int n = arr.length;

        // Call the countWaysToMakeChange function and print the result
        System.out.println("The total number of ways is " + countWaysToMakeChange(arr, n,
```

```
    target));
        }
    }
```

## Explanation:

1. **Initialization**:
    - `long dp[][] = new long[n][T + 1];`: Creates a 2D array `dp` to store the number of ways to make each target sum from `0` to `T` using elements from the array.
    - The outer loop initializes the base case for the first element. If the target `i` is divisible by `arr[0]`, there's one way to make the sum using only `arr[0]`.

2. **Dynamic Programming Table Fill**:
    - The outer loop iterates over each element in the array starting from the second element.
    - The inner loop iterates over each possible target sum from `0` to `T`.
    - `notTaken = dp[ind - 1][target];`: This calculates the number of ways to make the target sum without including the current element.
    - `taken = 0; if (arr[ind] <= target) taken = dp[ind][target - arr[ind]];`: This calculates the number of ways to make the target sum by including the current element (if it does not exceed the target).
    - `dp[ind][target] = notTaken + taken;`: The total number of ways to make the target sum is the sum of ways by not taking and taking the current element.

3. **Result**:
    - `return dp[n - 1][T];`: The final answer is stored in `dp[n - 1][T]`, which represents the total number of ways to make the target sum `T` using all elements in the array.

This code follows a clear dynamic programming approach to count the number of ways to make a target sum with unlimited usage of each coin.

Great question! The way we initialize the DP table for the "Minimum Coins" problem and the "Count Ways to Make Change" problem is different because the nature of these problems is different.

## Minimum Coins Problem:

**Objective:** Find the minimum number of coins needed to make a target sum.

**Initialization:**

- We initialize `dp[0][i]` with the minimum number of coins needed to make sum `i` using only the first coin.
- If the target sum `i` is divisible by `arr[0]`, then we can use `i / arr[0]` coins of `arr[0]`.

- Otherwise, if `i` is not divisible by `arr[0]`, it is impossible to form the sum using only this coin, so we set the value to a very large number (representing infinity or an impossible state).

**Code:**

```java
for (int i = 0; i <= T; i++) {
    if (i % arr[0] == 0)
        dp[0][i] = i / arr[0]; // Minimum coins needed when only using the first coin
    else
        dp[0][i] = (int) Math.pow(10, 9); // Impossible to form the sum with only the first coin
}
```

## Count Ways to Make Change Problem:

**Objective:** Count the number of ways to make a target sum using given coins.

**Initialization:**

- We initialize `dp[0][i]` with the number of ways to make sum `i` using only the first coin.
- If the target sum `i` is divisible by `arr[0]`, then there is exactly one way to form the sum using only `arr[0]` coins.
- If `i` is not divisible by `arr[0]`, there is no way to form the sum using only the first coin, so we set it to 0.

**Code:**

```java
for (int i = 0; i <= T; i++) {
    if (i % arr[0] == 0)
        dp[0][i] = 1; // Only one way to form the sum using only the first coin
    else
        dp[0][i] = 0; // No way to form the sum using only the first coin
}
```

## Identification Patterns:

1. **Objective of the Problem**:
    - **Minimum Coins:** We are interested in the minimum count, so we initialize with a large number to represent impossibility.
    - **Count Ways:** We are interested in the number of ways, so we initialize with zero when it's impossible and with one when it’s possible using only the first element.
2. **Nature of the Problem**:

- **Minimum Coins:** The problem deals with finding minimum values, hence large number initialization helps in identifying and updating the minimum count.
- **Count Ways:** The problem deals with counting combinations, hence initializing with zero or one helps in counting ways accurately.

## Applying These Patterns to New Problems:

When faced with a new dynamic programming problem, consider the following:

1. **Define the State:**
   - Determine what `dp[i][j]` represents. For example, `dp[i][j]` might represent the minimum number of coins to make the sum `j` using the first `i` coins or the number of ways to make the sum `j` using the first `i` coins.
2. **Identify the Base Case:**
   - Think about the simplest possible scenario. How would you solve the problem if there was only one coin or one item? This helps in setting up the initial values in the DP table.
3. **Transition/Recurrence Relation:**
   - Determine how to transition from one state to another. For example, whether to include or exclude an element, how the current state depends on previous states, etc.
4. **Initialization:**
   - Based on the problem objective, initialize the DP table:
     - For problems focusing on minimum/maximum values, initialize with a large/small number.
     - For counting problems, initialize with zero (impossible states) or one (single way).

## Example:

For a problem where you need to find the maximum sum of subsets:

- `dp[i][j]` might represent the maximum sum achievable with the first `i` elements and sum `j`.
- Initialization: Set `dp[0][0] = 0` (zero sum with zero elements) and `dp[0][j] = -∞` for all `j > 0` (impossible to form positive sum with zero elements).

```java
for (int i = 0; i <= T; i++) {
    if (i == 0)
        dp[0][i] = 0; // Zero sum with zero elements
    else
        dp[0][i] = Integer.MIN_VALUE; // Impossible state for non-zero sums with zero
elements
}
```

Understanding the objective and nature of the problem helps determine how to initialize and fill the DP table appropriately.