CS 2500: Algorithms

Lecture 24: Dynamic Programming: 0/1 Knapsack Problem

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

November 7, 2024

0/1 Knapsack Problem

We are given a number of objects and a knapsack.

- We suppose that the objects **may not be broken into smaller pieces**, so we may either take an object or leave it behind.
- Let $i = 1, 2, \ldots, n$ denote the objects.
- Each object i has:
  - a positive weight w_i
  - a positive value v_i
- The knapsack has a weight capacity W.
- Goal: Fill the knapsack in a way that maximizes the **total value of the included objects**.

Let $x_i = 0$ if we do not take object i, and $x_i = 1$ if we include object i.

0/1 Knapsack Problem

Mathematical Formulation

$$\text{maximize} \quad \sum_{i=1}^{n} x_i v_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} x_i w_i \leq W$$

where:

- $v_i > 0$ and $w_i > 0$
- $x_i \in \{0, 1\}$ for $1 \leq i \leq n$

Constraints:

- v_i and w_i are constants on the instance
- x_i are variables in the solution.

Objective: Maximize the total value without exceeding the weight capacity W.

0/1 Knapsack Problem

Greedy Algorithm

- The greedy algorithm suggests choosing objects in order of decreasing $\frac{v_i}{w_i}$ (value per unit weight).
- However, the greedy approach fails when objects cannot be broken.
- Example:
  - Suppose we have three objects:
    - Object 1: weight $= 6$, value $= 8$
    - Object 2: weight $= 5$, value $= 5$
    - Object 3: weight $= 5$, value $= 5$
  - Knapsack capacity $= 10$.
  - Greedy algorithm would pick object 1 (highest $\frac{v}{w}$), but this leaves no room for other objects.
  - Optimal solution: pick objects 2 and 3, with total value of 10.

Dynamic Programming: Two Main Approaches

- Dynamic programming can be implemented using two main approaches:
  1. **Top-Down with Memoization**
  2. **Bottom-Up**
- Both approaches use a table to store intermediate results, but differ in how the table is filled and the order in which subproblems are solved.

Note: The technical term "memoization" is not a misspelling of "memorization". The word "memoization" comes from "memo" since the technique consists of recording a value to be looked up later.

Order of Filling the Table

Top-Down with Memoization:

- Starts from the "top" (i.e., the original problem) and breaks it down into smaller subproblems recursively.
- For each subproblem, checks if the solution already exists in the table.
- If the solution is not in the table, solves the subproblem and stores the result in the table.
- Recursive calls continue until reaching the base cases, and results are stored ("memoized") as they are computed.

Bottom-Up:

- Begins from the "bottom" by solving the smallest subproblems first and storing their results in the table.
- Uses these stored results to solve larger subproblems iteratively, building up to the solution for the original problem.
- Systematically fills all entries in the table from smallest to largest subproblems.

Implementation Style

Top-Down with Memoization:

- Uses recursion with memoization.
- Each recursive call may lead to further recursive calls. Once a subproblem is solved, its result is stored to avoid redundant computation.
- Has a natural recursive structure, making the logic easier to understand for problems that are inherently recursive.

Bottom-Up:

- Uses an iterative approach with loops to fill the table from the smallest subproblems up to the largest.
- Implemented with explicit loops rather than recursive calls, reducing the overhead associated with recursion.
- Builds the solution in a structured, iterative manner, which can be more efficient in practice.

Computation Path

Top-Down with Memoization:

- Only the necessary subproblems are computed, depending on the recursion path taken to solve the main problem.
- In some cases, not all entries in the table are filled, as only the subproblems needed to reach the solution are computed.

Bottom-Up:

- Computes all possible subproblems in a systematic order.
- Every entry in the table is usually filled, even if not all of them are necessary to compute the final result.
- This approach ensures all dependencies are solved in advance, as the solution builds from the smallest subproblems.

Efficiency and Overhead

Top-Down with Memoization:

- Can have more overhead due to recursive calls, especially if the problem has deep recursion.
- May save some work by only computing the necessary subproblems, which can be efficient in some cases.

Bottom-Up:

- Typically has less overhead, as it avoids recursion and fills the table directly with iterative loops.
- This approach can lead to better performance in practice, especially in languages where recursion is costly.
- More suitable for problems where all subproblems need to be computed systematically.

Top-Down vs Bottom-Up: Summary

Top-Down with Memoization:

- Starts with the original problem and breaks it down recursively.
- Uses memoization to store results of subproblems.
- Only solves necessary subproblems based on recursion path.

Bottom-Up:

- Starts with the smallest subproblems and builds up iteratively.
- Systematically fills all table entries, ensuring all dependencies are solved.
- Avoids recursion overhead, typically faster in practice.

0/1 Knapsack: Top-Down Approach with Memoization

Steps for Top-Down Approach:

1. Start with the original problem (e.g., max value with all items and full weight capacity).
2. Recursively break down the problem:
   - If item i is not included, recursively compute the solution for $i-1$ items with the same capacity.
   - If item i is included, recursively compute the solution for $i-1$ items with reduced capacity $W - w_i$.
3. Store results in a table (memoization) as each subproblem is solved.
4. Retrieve results from the table when the same subproblem is encountered again, avoiding redundant computation.

> **Note:** This recursive approach does not require solving every subproblem in the table; only the subproblems reached by the recursion path are solved.

0/1 Knapsack: Top-Down Approach with Memoization

Let Knapsack(i, w) represent the max value for items 1 to i with weight w:

- Recursive formula:

$$\text{Knapsack}(i, w) = \begin{cases} \text{Knapsack}(i-1, w) & \text{if } w_i > w \\ \max(\text{Knapsack}(i-1, w), v_i + \text{Knapsack}(i-1, w-w_i)) & \text{if } w_i \leq w \end{cases}$$

- Results are stored in a table as subproblems are computed, avoiding redundant recursion.

0/1 Knapsack: Top-Down Approach with Memoization

Algorithm 2 0/1 Knapsack Problem - Top-Down with Memoization

Require: n: number of items, W: maximum weight capacity of the knapsack
Require: $w[i]$: weight of item i, $v[i]$: value of item i for $i = 1, \ldots, n$
Ensure: Maximum value achievable with weight limit W

Initialize a 2D memoization array $V[0 \ldots n][0 \ldots W]$ with all values set to -1 ▷ indicates uncomputed subproblems
function KNAPSACK(i, w)
    if $i = 0$ or $w = 0$ **then**
        return 0 ▷ Base case: no items or zero capacity yields zero value
    end if
    if $V[i][w] \neq -1$ **then**
        return $V[i][w]$ ▷ Return already computed value
    end if
    if $w_i \leq w$ **then**
        $V[i][w] \leftarrow \max(\text{Knapsack}(i-1, w), \text{Knapsack}(i-1, w-w_i) + v_i)$
    else
        $V[i][w] \leftarrow \text{Knapsack}(i-1, w)$
    end if
    return $V[i][w]$
end function
Compute the solution: Call KNAPSACK(n, W) to fill the memoization table and get the maximum value **return** $V[n][W]$

0/1 Knapsack: Bottom-Up Approach

Steps for Bottom-Up Approach:

1. Initialize a table (e.g., 2D array) where each entry represents a subproblem (e.g., max value achievable with a certain number of items and a certain weight capacity).
2. Fill the table iteratively, row by row or column by column, starting from the smallest subproblems (e.g., 0 items or 0 weight).
3. Use the already computed subproblem results to solve larger subproblems in each step.
4. The final entry in the table gives the answer to the original problem.

Note: This approach systematically fills all entries in the table, even if not all are necessary to compute the final result.

0/1 Knapsack: Bottom-Up Approach

- Define a table $V[i][w]$ where:
  - i: The number of items considered (from 1 to n).
  - w: The weight limit (from 0 to W).
  - $V[i][w]$: The maximum value achievable with items $1, \ldots, i$ and weight capacity w.
- Recurrence relation:

$$V[i][w] = \begin{cases} V[i-1][w] & \text{if } w_i > w \\ \max(V[i-1][w], V[i-1][w-w_i] + v_i) & \text{if } w_i \leq w \end{cases}$$

- Boundary conditions:
  - $V[0][w] = 0$ for all w (no items to include).
  - $V[i][0] = 0$ for all i (zero weight capacity).

0/1 Knapsack: Bottom-Up Approach

Algorithm 1 0/1 Knapsack Problem - Bottom-Up Approach

Require: n: number of items, W: maximum weight capacity of the knapsack
Require: $w[i]$: weight of item i, $v[i]$: value of item i for $i = 1, \ldots, n$
Ensure: Maximum value achievable with weight limit W

Initialize a 2D array $V[0 \ldots n][0 \ldots W]$ where $V[i][w]$ represents the maximum value achievable with the first i items and weight limit w
for $i = 0$ to n **do**
    $V[i][0] \leftarrow 0$ {0 capacity results in 0 value}
end for
for $w = 0$ to W **do**
    $V[0][w] \leftarrow 0$ {0 items result in 0 value for any capacity}
end for
for $i = 1$ to n **do**
    for $w = 1$ to W **do**
        if $w_i \leq w$ **then**
            $V[i][w] \leftarrow \max(V[i-1][w], V[i-1][w - w_i] + v_i)$
        else
            $V[i][w] \leftarrow V[i-1][w]$
        end if
    end for
end for
return $V[n][W]$ {The maximum value achievable with all items and capacity W} =0

Example:

- Objects: weights = 1, 2, 5, 6, 7; values = 1, 6, 18, 22, 28.
- Knapsack capacity = 11.
- Dynamic programming table shows maximum values at different weight capacities.

Weight limit:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1, v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2, v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5, v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6, v_4 = 22$	0	1	6	7	7	18	22	23	28	29	29	40
$w_5 = 7, v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

Figure: The 0/1 Knapsack Problem using Dynamic Programming.

0/1 Knapsack: Bottom-Up Approach

Row 1: $w_1 = 1, v_1 = 1$

- For weight $w = 0$: $V[1][0] = 0$ (knapsack has zero capacity).
- For weight $w \geq 1$: Item 1 fits, so $V[1][w] = 1$.

Row 1: $0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1$

0/1 Knapsack: Bottom-Up Approach

Row 2: $w_2 = 2, v_2 = 6$

- For $w = 0$: $V[2][0] = 0$.
- For $w = 1$: $V[2][1] = 1$ (only item 1 fits).
- For $w = 2$: Use item 2 alone, $V[2][2] = 6$.
- For $w = 3$: Combine item 1 and item 2, $V[2][3] = 7$.
- For $w \geq 4$: Value remains 7.

Row 2: $0, 1, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7$

0/1 Knapsack: Bottom-Up Approach

Row 3: $w_3 = 5, v_3 = 18$

- For $w \leq 4$: Item 3 cannot fit, $V[3][w] = V[2][w]$.
- For $w = 5$: Use item 3 alone, $V[3][5] = 18$.
- For $w = 6$: Use item 3 and item 1, $V[3][6] = 19$.
- For $w = 7$: Use item 3 and item 2, $V[3][7] = 24$.
- For $w \geq 8$: Maximum value is 25.

Row 3: $0, 1, 6, 7, 7, 18, 19, 24, 25, 25, 25, 25$

0/1 Knapsack: Bottom-Up Approach

Row 4: $w_4 = 6, v_4 = 22$

- For $w \leq 5$: Item 4 cannot fit, $V[4][w] = V[3][w]$.
- For $w = 6$: Use item 4 alone, $V[4][6] = 22$.
- For $w = 7$: Use item 4 and item 1, $V[4][7] = 23$.
- For $w = 8$: Use item 4 and item 2, $V[4][8] = 28$.
- For $w = 9$: Maximum value is 29.
- For $w = 11$: Use items 4 and 3, giving $V[4][11] = 40$.

Row 4: $0, 1, 6, 7, 7, 18, 22, 23, 28, 29, 29, 40$

Row 5: $w_5 = 7, v_5 = 28$

- For $w \leq 6$: Item 5 cannot fit, $V[5][w] = V[4][w]$.
- For $w = 7$: Use item 5 alone, $V[5][7] = 28$.
- For $w = 8$: Use item 5 and item 1, $V[5][8] = 29$.
- For $w = 9$: Use item 5 and item 2, $V[5][9] = 34$.
- For $w = 10$: Maximum value is 35.
- For $w = 11$: Use items 5 and 3 or items 5 and 4, $V[5][11] = 40$.

Row 5: $0, 1, 6, 7, 7, 18, 22, 28, 29, 34, 35, 40$

Optimal Solution Traceback

- Using the table, trace back to find the composition of the optimal load.
- Example:
    - Start from $V[5, 11]$, check previous cells to identify items included in the optimal solution.
    - Optimal solution consists of objects 3 and 4.
    - Total value $= 40$.

0/1 Knapsack: Bottom-Up Approach

Algorithm Complexity

- Time Complexity: $O(nW)$
  - n: Number of items.
  - W: Knapsack capacity.
- Space Complexity: $O(nW)$ for storing the table V.
- Efficient for cases where both n and W are not too large.

Subset Sum Problem

- Goal: Determine if there exists a subset of a given set of integers that sums up to a target value S.
- This problem can be solved efficiently using DP.
- It shares similarities with the 0/1 Knapsack Problem.

Subset Sum Problem

Dynamic Programming Approach: To solve the Subset Sum Problem using DP, we define a DP table where:

- DP[i][j]: Boolean value indicating whether a subset of the first i elements can sum up to j.
- DP[i][j] $=$ True if there exists a subset that sums up to j.
- DP[i][j] $=$ False otherwise.

Subset Sum Problem

Similarity to the 0/1 Knapsack Problem

- **Decision Problem vs. Optimization Problem**:
  - Subset Sum is a decision problem (we only care if a solution exists).
  - 0/1 Knapsack is an optimization problem (we want the maximum value).
- **Structure of Choices**:
  - In both problems, for each element, we can either include or exclude it.

Subset Sum Problem

Recurrence Relation Comparison

- **Subset Sum**:

$$\mathrm{DP}[i][j] = \mathrm{DP}[i-1][j] \text{ OR } \mathrm{DP}[i-1][j-a_{i-1}]$$

- **0/1 Knapsack**:

$$\mathrm{DP}[i][j] = \max(\mathrm{DP}[i-1][j], \mathrm{DP}[i-1][j-w_i]+v_i)$$

- In Subset Sum, we use boolean OR; in 0/1 Knapsack, we use max.

DP Table Initialization

1. **Base Case:** $DP[0][0] = \text{True}$. With zero elements, we can achieve a sum of 0 by taking an empty subset.
2. **First Row:** For $j > 0$, $DP[0][j] = \text{False}$: With zero elements, no non-zero sum is achievable.

Filling the DP Table: For each element i (where $1 \leq i \leq n$) and each possible sum j (where $0 \leq j \leq S$):

- If a_{i-1} is greater than j: Exclude it, so

$$\text{DP}[i][j] = \text{DP}[i-1][j]$$

- If $a_{i-1} \leq j$: We have two choices:

$$\text{DP}[i][j] = \text{DP}[i-1][j] \text{ OR } \text{DP}[i-1][j - a_{i-1}]$$

Explanation of the Choices

- DP[$i-1$][j]: The subset sum j can be achieved without including a_{i-1}.
- DP[$i-1$][$j-a_{i-1}$]: The subset sum j can be achieved by including a_{i-1}, provided that a subset summing to $j-a_{i-1}$ exists among the first $i-1$ elements.

Subset Sum Problem

Result of the DP Table: The solution to the problem will be found in the cell DP$[n][S]$:

- If DP$[n][S]$ = True, there exists a subset of the array that sums to S.
- If DP$[n][S]$ = False, no such subset exists.

Subset Sum Problem

Example:

- Array: $\{3, 34, 4, 12, 5, 2\}$
- Target Sum: 9

We fill the DP table using the rules discussed and check DP[6][9] to see if a subset with sum 9 exists.

Subset Sum Problem

- Given Array: $\{3, 34, 4, 12, 5, 2\}$
- Target Sum: 9

i	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$	$j=8$	$j=9$
0 elements	T	F	F	F	F	F	F	F	F	F
3	T	F	F	T	F	F	F	F	F	F
3, 34	T	F	F	T	F	F	F	F	F	F
3, 34, 4	T	F	F	T	T	F	F	T	F	F
3, 34, 4, 12	T	F	F	T	T	F	F	T	F	F
3, 34, 4, 12, 5	T	F	F	T	T	T	F	T	T	T
3, 34, 4, 12, 5, 2	T	F	T	T	T	T	T	T	T	T

Figure: DP Table for Array $\{3, 34, 4, 12, 5, 2\}$ and Target Sum 9

Subset Sum Problem

Row 0: 0 Elements

- With 0 elements, the only achievable sum is 0.
- Therefore, DP[0][0] $=$ True, and all other entries are **False**.

Row 0: T, F, F, F, F, F, F, F, F, F

Subset Sum Problem

Row 1: First Element = 3

- For $j = 3$: We can achieve a sum of 3 by taking only the element 3, so DP[1][3] = True.
- For other values of $j > 3$: No subset exists that sums to those values with just the element 3.

Row 1: T, F, F, T, F, F, F, F, F, F

Subset Sum Problem

Row 2: Elements $= \{3, 34\}$

- Adding 34 doesn't help us achieve any new sums below 34.
- Therefore, this row remains the same as Row 1.

Row 2: T, F, F, T, F, F, F, F, F, F

Subset Sum Problem

Row 3: Elements = $\{3, 34, 4\}$

- For $j = 4$: We can achieve a sum of 4 by using only the element 4, so DP[3][4] = True.
- For $j = 7$: We can achieve a sum of 7 by combining elements 3 and 4, so DP[3][7] = True.

Row 3: T, F, F, T, T, F, F, T, F, F

Row 4: Elements $= \{3, 34, 4, 12\}$

- Adding 12 doesn't allow us to achieve any new sums below 12.
- Therefore, this row remains the same as Row 3.

Row 4: T, F, F, T, T, F, F, T, F, F

Subset Sum Problem

Row 5: Elements = $\{3, 34, 4, 12, 5\}$

- For $j = 5$: We can achieve a sum of 5 by using only the element 5, so DP[5][5] = True.
- For $j = 8$: We can achieve a sum of 8 by combining elements 3 and 5, so DP[5][8] = True.
- For $j = 9$: We can achieve a sum of 9 by combining elements 4 and 5, so DP[5][9] = True.

Row 5: T, F, F, T, T, T, F, T, T, T

Subset Sum Problem

Row 6: Elements = $\{3, 34, 4, 12, 5, 2\}$

- For $j = 2$: We can achieve a sum of 2 by using only the element 2, so DP[6][2] $=$ True.
- For $j = 6$: We can achieve a sum of 6 by combining elements 4 and 2, so DP[6][6] $=$ True.
- For $j = 7$: We can achieve a sum of 7, so this remains True.
- For $j = 8$ and $j = 9$: These values remain True from previous calculations.

Row 6: T, F, T, T, T, T, T, T, T, T

Steps to Trace Back and Find the Subset

- To identify the subset, trace back through the DP table:
    1. Start from DP[n][S].
    2. Compare each DP[i][j] with DP[$i-1$][j].
    3. Record any element that changes DP[i][j] from DP[$i-1$][j].

Subset Sum Problem

Step 1: Start from DP$[n][S]$

- Begin at DP[6][9], representing using the first 6 elements to achieve a sum of 9.
- Our goal is to trace back through the table to identify which elements contribute to this sum.

Subset Sum Problem

Step 2: Check Each Element's Contribution

- For each DP[i][j]:
    - If DP[i][j] $=$ DP[$i-1$][j], the i-th element was not included. Move up to DP[$i-1$][j].
    - If DP[i][j] $\neq$ DP[$i-1$][j], the i-th element was included. Record it, subtract its value from j, and move to DP[$i-1$][$j-a_{i-1}$].

Subset Sum Problem

Example Traceback: Starting Point

- **Starting at DP**[6][9]:
  - DP[6][9] = True and DP[5][9] = True.
  - Therefore, the 6th element (2) was not needed for the sum. Move to DP[5][9].

Example Traceback: Moving to DP[5][9]

- **Check DP**[5][9] **and DP**[4][9]:
  - DP[5][9] = True but DP[4][9] = False.
  - This means the 5th element (5) was included in the subset.
  - Record 5 as part of the subset and update $j = 9 - 5 = 4$.
  - Move to DP[4][4].

Example Traceback: Moving to DP[4][4]

- **Check DP**[4][4] **and DP**[3][4]:
  - DP[4][4] = True and DP[3][4] = True.
  - This means the 4th element (12) was not included.
  - Move to DP[3][4].

Subset Sum Problem

Example Traceback: Moving to DP[3][4]

- **Check DP**[3][4] **and DP**[2][4]:
    - DP[3][4] = True but DP[2][4] = False.
    - This indicates the 3rd element (4) was included.
    - Record 4 as part of the subset and update $j = 4 - 4 = 0$.

Subset Sum Problem

Stop Condition

- Since $j = 0$, we have identified all elements in the subset that sum to the target.
- Solution subset: $\{5, 4\}$.

Subset Sum Problem

Summary:

- The Subset Sum Problem is essentially a simplified version of the 0/1 Knapsack Problem.
- Subset Sum corresponds to a knapsack problem where each item has a "value" equal to its weight.
- Both share structural similarities in terms of choices and DP table setup.

Equal Sum Partition Problem

Problem Statement

- Given a set of n positive integers $\{a_1, a_2, \ldots, a_n\}$.
- Determine if it is possible to partition the set into two subsets with equal sums.

Why Total Sum Must Be Even

- For two subsets to have equal sums, the total sum of the array must be even.
- Let Total Sum be the sum of all elements in the array.
- If the array can be partitioned into two equal-sum subsets, each subset must sum to:

$$\text{target} = \frac{\text{Total Sum}}{2}$$

- If Total Sum is odd, dividing by 2 results in a non-integer, making an equal partition impossible.

Equal Sum Partition Problem

Example to Illustrate the Even Sum Requirement

- Example 1: Array $= \{1, 5, 11, 5\}$
  - Total Sum $= 1 + 5 + 11 + 5 = 22$ (even).
  - Possible to split into subsets that sum to 11.
- Example 2: Array $= \{1, 2, 4\}$
  - Total Sum $= 1 + 2 + 4 = 7$ (odd).
  - Equal partition is impossible because half of 7 is 3.5, not an integer.
- Conclusion: If Total Sum is odd, return False immediately.

Equal Sum Partition Problem

Reducing to a Subset Sum Problem

- If Total Sum is even, the problem reduces to finding a subset that sums to:

$$\text{target} = \frac{\text{Total Sum}}{2}$$

- This is now a **Subset Sum Problem** where the goal is to check if any subset can sum up to target.

Equal Sum Partition Problem

Summary

- If the total sum is odd, an equal partition is impossible.
- If the total sum is even, reduce the problem to finding a subset sum equal to half the total sum.
- Solve using a DP table to check if a subset with the target sum exists.
- Efficiently determines if an equal partition is possible.

Count of Subsets with a Given Sum Problem

Problem Statement

- Given a set of n positive integers $\{a_1, a_2, \ldots, a_n\}$.
- A target sum S.
- Objective: Count the number of subsets in the set that sum up to exactly S.

Count of Subsets with a Given Sum Problem

How is this a Variation of the Subset Sum Problem?

- This problem is a variation of the **Subset Sum Problem**, where we are not just interested in checking if a subset exists, but in counting all possible subsets that sum to the target S.

Count of Subsets with a Given Sum Problem

Key Differences and Similarities with Subset Sum

1. **Problem Objective**:
   - In the **Subset Sum Problem**, we simply check whether there exists at least one subset that sums up to a given target.
   - In the **Count of Subsets with a Given Sum Problem**, the objective is to count all possible subsets that sum to the target, rather than just determining existence.

2. **DP Table Structure**:
   - Both problems use a dynamic programming (DP) table to keep track of achievable sums up to the target.
   - In the **Subset Sum Problem**, the DP table typically holds boolean values (`True` or `False`) indicating whether a specific sum can be achieved.
   - In the **Count of Subsets with a Given Sum Problem**, the DP table holds integer counts, with each cell representing the number of ways to achieve a particular sum with the first i elements.

Count of Subsets with a Given Sum Problem

3. **Filling the DP Table**:
    - The structure of the DP table and the base cases are very similar in both problems.
    - For each element, you decide whether to include or exclude it. The primary difference lies in how you update the table:
        - In **Subset Sum**, you use a logical `OR` to determine if any subset can achieve the target sum.
        - In **Count of Subsets**, you use addition to accumulate the counts of subsets that can form the target sum.

Count of Subsets with a Given Sum Problem

4. **Recurrence Relation**:
   - In **Subset Sum**:

$$\text{DP}[i][j] = \text{DP}[i-1][j] \text{ OR } \text{DP}[i-1][j-a_{i-1}]$$

   - In **Count of Subsets**:

$$\text{DP}[i][j] = \text{DP}[i-1][j] + \text{DP}[i-1][j-a_{i-1}]$$

   - Here, the + operation in the count problem replaces the `OR` operation in the subset sum problem.

5. **Final Result Extraction**:
   - In **Subset Sum**, you simply check if the target sum is achievable by examining if $\text{DP}[n][S] = \text{True}$.
   - In **Count of Subsets**, you directly retrieve the number of subsets with sum equal to S by reading the value $\text{DP}[n][S]$.

Summary

- The **Count of Subsets with a Given Sum Problem** can be thought of as an **extension of the Subset Sum Problem**.
- Instead of verifying the existence of a subset, you count all possible subsets that sum to a given target.
- This variation leverages a similar DP setup but with a focus on counting configurations, making it a natural extension of subset sum concepts.