Knapsack Problem Java Solution

Explanation and Solution for the Knapsack Problem

The problem you provided is the classic 0/1 Knapsack Problem. The goal is to maximize the total value of items that can be placed in a knapsack of a given capacity. Each item has a weight and a value, and you can either include an item in the knapsack or exclude it (hence the name 0/1).

Here's a step-by-step breakdown of the dynamic programming approach used in the code:

1. **Initialization**:
   - We create a 2D array `K` where `K[i][w]` represents the maximum value that can be obtained using the first `i` items and a knapsack capacity of `w`.
2. **Base Case**:
   - If there are no items (`i == 0`) or the capacity of the knapsack is zero (`w == 0`), then the maximum value is 0. This is initialized in the table as `K[i][w] = 0`.
3. **Recursive Case**:
   - If the weight of the `i-th` item is less than or equal to the current capacity `w`, we have two choices:
     1. Include the item and add its value to the result of the remaining capacity: `val[i-1] + K[i-1][w-wt[i-1]]`.
     2. Exclude the item and take the value without it: `K[i-1][w]`.
   - We take the maximum of these two choices.
   - If the weight of the `i-th` item is more than the current capacity `w`, we cannot include it, so we take the value without it: `K[i-1][w]`.
4. **Result**:
   - The value at `K[n][W]` will be the maximum value that can be achieved with `n` items and knapsack capacity `W`.

Java Code with Detailed Comments

```java
class Solution {
    /**
     * Function to return max value that can be put in knapsack of capacity W.
```

```
     *
     * @param W    - Maximum weight capacity of the knapsack
     * @param wt   - Array of weights of the items
     * @param val  - Array of values of the items
     * @param n    - Number of items
     * @return     - Maximum value that can be put in the knapsack
     */
    static int knapSack(int W, int wt[], int val[], int n) {
        // Create a 2D array K where K[i][w] will store the maximum value
        // that can be obtained with i items and capacity w.
        int[][] K = new int[n + 1][W + 1];

        // Build the table K[][] in a bottom-up manner
        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= W; w++) {
                // Base case: If there are no items or the capacity is 0, max value is 0
                if (i == 0 || w == 0) {
                    K[i][w] = 0;
                }
                // If the weight of the current item is less than or equal to the current
capacity
                // We have the option to include it or exclude it
                else if (wt[i - 1] <= w) {
                    K[i][w] = Math.max(
                        val[i - 1] + K[i - 1][w - wt[i - 1]], // Include the item
                        K[i - 1][w] // Exclude the item
                    );
                }
                // If the weight of the current item is more than the current capacity
                // We cannot include it, so we take the value without it
                else {
                    K[i][w] = K[i - 1][w];
                }
            }
        }
        // The result is stored in K[n][W], which is the maximum value for n items and
capacity W
        return K[n][W];
    }
}
```

Dry Run

Let's perform a dry run for a small example to understand the logic better:

- **Input**: `W = 4`, `wt = [1, 3, 4, 5]`, `val = [1, 4, 5, 7]`, `n = 4`

Table Construction (`K`):

```
css
```

```
          w = 0  1  2  3  4
i = 0   [ 0, 0, 0, 0, 0 ]
i = 1   [ 0, 1, 1, 1, 1 ]
i = 2   [ 0, 1, 1, 4, 5 ]
i = 3   [ 0, 1, 1, 4, 5 ]
i = 4   [ 0, 1, 1, 4, 5 ]
```

- **Result**: The maximum value that can be put in the knapsack of capacity 4 is 5.

Common Mistakes and Edge Cases

1. **Off-by-One Errors**:
   - Ensure the indexing is correct, especially when accessing the item arrays (`wt` and `val`). Remember `wt[i-1]` and `val[i-1]` because `i` starts from 1 in the table but 0 in the arrays.
2. **Initialization**:
   - Properly initialize the base cases in the table (`K[i][0] = 0` and `K[0][w] = 0`).
3. **Edge Cases**:
   - No items (`n = 0`): The result should be 0.
   - Knapsack capacity 0 (`W = 0`): The result should be 0.
   - All items are too heavy for the knapsack capacity.

Understanding and implementing the 0/1 Knapsack Problem using dynamic programming helps build a strong foundation for solving more complex problems and preparing for technical interviews.

Sure! Let's explore both the brute-force recursive approach and the optimal dynamic programming (DP) approach for the 0/1 Knapsack problem.

Brute Force Recursive Approach

The brute force approach explores all possible subsets of items to find the subset with the maximum total value without exceeding the knapsack's capacity. This approach has an exponential time complexity, but it helps in understanding the problem's structure.

Algorithm

1. If no items are left or the capacity of the knapsack is 0, return 0.
2. For each item, check if its weight is less than or equal to the current capacity.
   - If yes, compute the maximum value by either including the item or excluding it.
   - If no, move to the next item without including it.
3. Return the maximum of the above choices.

Java Code

```java
class Solution {
    /**
     * Brute force recursive function to find the maximum value for knapsack capacity.
     *
     * @param W    - Maximum weight capacity of the knapsack
```

```
     * @param wt   - Array of weights of the items
     * @param val  - Array of values of the items
     * @param n    - Number of items
     * @return     - Maximum value that can be put in the knapsack
     */
    static int knapSackRecursive(int W, int wt[], int val[], int n) {
        // Base case: No items left or capacity is 0
        if (n == 0 || W == 0) {
            return 0;
        }

        // If the weight of the current item is more than the knapsack capacity, skip it
        if (wt[n - 1] > W) {
            return knapSackRecursive(W, wt, val, n - 1);
        } else {
            // Return the maximum value by including or excluding the current item
            return Math.max(
                val[n - 1] + knapSackRecursive(W - wt[n - 1], wt, val, n - 1), // Include
the item
                knapSackRecursive(W, wt, val, n - 1) // Exclude the item
            );
        }
    }
}
```

Dynamic Programming (DP) Approach

The DP approach optimizes the recursive solution by storing intermediate results, avoiding redundant calculations. This approach has a time complexity of $O(n \cdot W)$.

Algorithm

1. Initialize a 2D array `K` with dimensions $(n+1) \times (W+1)$ to store the maximum value that can be obtained using the first `i` items and capacity `w`.
2. Iterate over all items and capacities, filling the table based on the choice of including or excluding each item.
3. Return the value at `K[n][W]`.

Java Code

java

```
class Solution {
    /**
     * Dynamic Programming function to find the maximum value for knapsack capacity.
     *
     * @param W    - Maximum weight capacity of the knapsack
     * @param wt   - Array of weights of the items
     * @param val  - Array of values of the items
     * @param n    - Number of items
     * @return     - Maximum value that can be put in the knapsack
     */
    static int knapSackDP(int W, int wt[], int val[], int n) {
        // Create a 2D array K where K[i][w] will store the maximum value
        // that can be obtained with i items and capacity w.
        int[][] K = new int[n + 1][W + 1];

        // Build the table K[][] in a bottom-up manner
        for (int i = 0; i <= n; i++) {
```

```
            for (int w = 0; w <= W; w++) {
                // Base case: If there are no items or the capacity is 0, max value is 0
                if (i == 0 || w == 0) {
                    K[i][w] = 0;
                }
                // If the weight of the current item is less than or equal to the current
capacity
                // We have the option to include it or exclude it
                else if (wt[i - 1] <= w) {
                    K[i][w] = Math.max(
                        val[i - 1] + K[i - 1][w - wt[i - 1]], // Include the item
                        K[i - 1][w] // Exclude the item
                    );
                }
                // If the weight of the current item is more than the current capacity
                // We cannot include it, so we take the value without it
                else {
                    K[i][w] = K[i - 1][w];
                }
            }
        }
        // The result is stored in K[n][W], which is the maximum value for n items and
capacity W
        return K[n][W];
    }
}
```

Summary of Approaches

- **Brute Force Recursive Approach**:
  - **Pros**: Simple and straightforward.
  - **Cons**: Exponential time complexity, impractical for large inputs.
- **Dynamic Programming Approach**:
  - **Pros**: Efficient with polynomial time complexity $O(n \cdot W)$, suitable for larger inputs.
  - **Cons**: Requires additional space for the DP table.

Understanding both approaches provides a comprehensive grasp of the problem, useful for building the logic to tackle similar problems in competitive programming and technical interviews.

Sure! Let's break down what gets stored at every index in both the brute force recursive approach and the dynamic programming (DP) approach for the 0/1 Knapsack problem.

Brute Force Recursive Approach

In the brute force recursive approach, we don't explicitly store values in an array or table. Instead, the maximum value is computed through recursive calls. Here's what happens at each call:

1. **Function Call**: `knapSackRecursive(W, wt, val, n)`

- **Parameters**:
  - `W`: Remaining capacity of the knapsack.
  - `wt`: Array of weights of the items.
  - `val`: Array of values of the items.
  - `n`: Number of items considered so far.
- **Returns**: Maximum value that can be achieved with `n` items and knapsack capacity `W`.

2. **Recursive Case**:

- If `wt[n-1] > W`, the item cannot be included, so we call `knapSackRecursive(W, wt, val, n-1)`.
- Otherwise, we consider two scenarios:
  - Include the item: `val[n-1] + knapSackRecursive(W - wt[n-1], wt, val, n-1)`.
  - Exclude the item: `knapSackRecursive(W, wt, val, n-1)`.

Dynamic Programming (DP) Approach

In the DP approach, we use a 2D table `K` where `K[i][w]` represents the maximum value that can be obtained using the first `i` items and a knapsack capacity of `w`.

Here's what gets stored at each index `K[i][w]`:

1. **Initialization**:

- `K[i][0] = 0` for all `i` (0 capacity).
- `K[0][w] = 0` for all `w` (0 items).

2. **Filling the Table**:

- For each item `i` (1 to `n`):
  - For each capacity `w` (1 to `W`):
    - **Base Case**: `if (i == 0 || w == 0) K[i][w] = 0`.
    - **If Item Can Be Included**: `if (wt[i-1] <= w) K[i][w] = Math.max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])`.
    - **If Item Cannot Be Included**: `else K[i][w] = K[i-1][w]`.

Detailed Explanation with Example

Let's use an example to clarify:

- **Input**: `W = 4`, `wt = [1, 3, 4, 5]`, `val = [1, 4, 5, 7]`, `n = 4`.

DP Table Construction (`K`):

```
css
```

```
          w = 0  1  2  3  4
i = 0   [ 0, 0, 0, 0, 0 ]
i = 1   [ 0, 1, 1, 1, 1 ]
```

```
i = 2   [ 0, 1, 1, 4, 5 ]
i = 3   [ 0, 1, 1, 4, 5 ]
i = 4   [ 0, 1, 1, 4, 5 ]
```

1. **Initialization**:
    - `K[0][w]` and `K[i][0]` are all initialized to 0.
2. **Filling the Table**:
    - **i = 1**:
        - **w = 1**: `wt[0] <= 1`, `K[1][1] = max(1 + K[0][0], K[0][1]) = 1`.
        - **w = 2**: `wt[0] <= 2`, `K[1][2] = max(1 + K[0][1], K[0][2]) = 1`.
        - **w = 3**: `wt[0] <= 3`, `K[1][3] = max(1 + K[0][2], K[0][3]) = 1`.
        - **w = 4**: `wt[0] <= 4`, `K[1][4] = max(1 + K[0][3], K[0][4]) = 1`.
    - **i = 2**:
        - **w = 1**: `wt[1] > 1`, `K[2][1] = K[1][1] = 1`.
        - **w = 2**: `wt[1] > 2`, `K[2][2] = K[1][2] = 1`.
        - **w = 3**: `wt[1] <= 3`, `K[2][3] = max(4 + K[1][0], K[1][3]) = 4`.
        - **w = 4**: `wt[1] <= 4`, `K[2][4] = max(4 + K[1][1], K[1][4]) = 5`.
    - **i = 3**:
        - **w = 1**: `wt[2] > 1`, `K[3][1] = K[2][1] = 1`.
        - **w = 2**: `wt[2] > 2`, `K[3][2] = K[2][2] = 1`.
        - **w = 3**: `wt[2] > 3`, `K[3][3] = K[2][3] = 4`.
        - **w = 4**: `wt[2] <= 4`, `K[3][4] = max(5 + K[2][0], K[2][4]) = 5`.
    - **i = 4**:
        - **w = 1**: `wt[3] > 1`, `K[4][1] = K[3][1] = 1`.
        - **w = 2**: `wt[3] > 2`, `K[4][2] = K[3][2] = 1`.
        - **w = 3**: `wt[3] > 3`, `K[4][3] = K[3][3] = 4`.
        - **w = 4**: `wt[3] > 4`, `K[4][4] = K[3][4] = 5`.

The final value at `K[n][W]` (i.e., `K[4][4]`) is 5, which is the maximum value for the given knapsack capacity.

Summary of Stored Values

- **Brute Force Recursive Approach**:
    - No explicit storage; values are computed through recursive calls and return the maximum value directly.
- **Dynamic Programming (DP) Approach**:
    - `K[i][w]` stores the maximum value that can be obtained with the first `i` items and capacity `w`.
    - Base cases are initialized to 0.
    - Values are updated based on whether the current item is included or excluded.