

Dynammmic programming codes dry run

Binary (0/1) knapsack problem to maximize the profit

Certainly! Let's dry run the provided pseudocode for the Knapsack problem, which is a classic algorithmic problem in the field of computer science and optimization. The function `KSP(W[], V[], N, M)` seems to be a dynamic programming solution for the 0/1 Knapsack problem, where `W[]` is the array of weights, `V[]` is the array of values, `N` is the number of items, and `M` is the maximum weight capacity of the knapsack.

Here's how the algorithm works step by step:

1. **Declaration:** A 2D array `R[0...N][0...M]` is declared to store the maximum value that can be achieved with the first `i` items and a weight limit of `j`.
2. **Initialization:** The first row and the first column of `R` are initialized to 0 because if there are no items or the weight limit is 0, the maximum value is 0.
3. **Computation:** The array `R` is filled using the following logic:
 - For each item `i` from 1 to `N`, and for each weight limit `j` from 1 to `M`:
 - If the weight of the current item `W[i]` is greater than `j`, the item cannot be included, so the value is the same as the maximum value without this item, which is `R[i-1][j]`.
 - If the weight of the current item `W[i]` is less than or equal to `j`, the item can be included, so the value is the maximum of:
 - The maximum value without this item (`R[i-1][j]`), and
 - The value of this item plus the maximum value of the remaining weight limit (`V[i] + R[i][j-W[i]]`).
4. **Final Result:** The function returns `R[N][M]`, which is the maximum value that can be achieved with `N` items and a weight limit of `M`.

To visualize this, let's assume we have `N = 3` items with weights `W[] = {1, 3, 4}` and values `V[] = {15, 20, 30}`, and the maximum weight capacity of the knapsack `M = 6`. The `R` matrix would be filled as follows:

```
R = [  
  [0, 0, 0, 0, 0, 0, 0], // Weight limit 0  
  [0, 15, 15, 15, 15, 15, 15], // Item 1 only  
  [0, 15, 15, 20, 35, 35, 35], // Item 1 and 2  
  [0, 15, 15, 20, 35, 35, 50]  // All items  
]
```

- The first row and column remain 0 after initialization.
- When item 1 is considered, it can be included for all weight limits from 1 to 6, so the first row is filled with the value of item 1, which is 15.

- When item 2 is considered, it can only be included for weight limits 3 and above. For weight limit 3, the value is 20 (item 2 only). For weight limits 4 to 6, the value is 35 (item 1 and 2).
- When item 3 is considered, it can only be included for weight limits 4 and above. For weight limit 4, the value remains 35 (item 1 and 2). For weight limit 6, the value is 50 (item 1 and 3).

The final result $R[N][M]$ or $R[3][6]$ is 50, which is the maximum value that can be achieved with the given items and weight limit.

I hope this dry run and visualization help you understand how the Knapsack problem is solved using dynamic programming. If you have any further questions or need more assistance, feel free to ask

Let's visualize the dynamic programming table R for the Knapsack problem step by step, highlighting the current values being considered and updated. We'll use the example with $N = 3$ items, weights $W[] = \{1, 3, 4\}$, values $V[] = \{15, 20, 30\}$, and maximum weight capacity $M = 6$.

Step 1: Initialization

```
R = [
  [0, 0, 0, 0, 0, 0, 0], // Weight limit 0
  [0, 0, 0, 0, 0, 0, 0], // Item 1
  [0, 0, 0, 0, 0, 0, 0], // Item 2
  [0, 0, 0, 0, 0, 0, 0]  // Item 3
]
```

Step 2: Consider Item 1 ($W[1] = 1, V[1] = 15$)

- For each weight limit j from 1 to 6, we can include Item 1 since $W[1] \leq j$.
- Update $R[1][j]$ to $V[1] + R[1][j-W[1]]$.

```
R = [
  [0, 0, 0, 0, 0, 0, 0], // Weight limit 0
  [0, **15**, **15**, **15**, **15**, **15**, **15**], // Item 1
  [0, 0, 0, 0, 0, 0, 0], // Item 2
  [0, 0, 0, 0, 0, 0, 0]  // Item 3
]
```

Step 3: Consider Item 2 ($W[2] = 3, V[2] = 20$)

- For weight limits $j < W[2]$, we cannot include Item 2, so $R[2][j] = R[1][j]$.
- For weight limits $j \geq W[2]$, we compare $R[1][j]$ and $V[2] + R[2][j-W[2]]$.

```
R = [
  [0, 0, 0, 0, 0, 0, 0], // Weight limit 0
  [0, 15, 15, 15, 15, 15, 15], // Item 1
  [0, 15, 15, **20**, **35**, **35**, **35**], // Item 2
  [0, 0, 0, 0, 0, 0, 0]  // Item 3
]
```

Step 4: Consider Item 3 ($W[3] = 4$, $V[3] = 30$)

- For weight limits $j < W[3]$, we cannot include Item 3, so $R[3][j] = R[2][j]$.
- For weight limits $j \geq W[3]$, we compare $R[2][j]$ and $V[3] + R[3][j-W[3]]$.

```
R = [  
  [0, 0, 0, 0, 0, 0, 0], // Weight limit 0  
  [0, 15, 15, 15, 15, 15, 15], // Item 1  
  [0, 15, 15, 20, 35, 35, 35], // Item 2  
  [0, 15, 15, 20, **35**, **35**, **50**] // Item 3  
]
```

In the final table, the highlighted values show the maximum value that can be achieved for each weight limit using the available items. The value $R[3][6] = 50$ is the maximum value that can be achieved with a weight limit of 6.

This visualization shows the decision-making process at each step, considering whether to include an item based on its weight and value, and updating the table accordingly. The highlighted values indicate the choices made by the algorithm at each step to achieve the maximum value. If you need further clarification or have more questions, feel free to ask

Coin Change Problem to return the amount with minimum number of coins.

Let's visualize the dynamic programming table R for the coin change problem step by step, highlighting the current values being considered and updated. We'll use the example with $\text{coins} = \{1, 2, 5\}$ and $\text{amount} = 11$.

Step 1: Initialization

```
R = [  
  [0, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX,  
  INT_MAX, INT_MAX, INT_MAX, INT_MAX], // Amount 0  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], // Coin 1  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], // Coin 2  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] // Coin 5  
]
```

Step 2: Consider Coin 1 (value = 1)

- For each amount j from 1 to 11, we can include Coin 1 since its value is 1.
- Update $R[1][j]$ to $1 + R[1][j - \text{coins}[0]]$.

```
R = [  
  [0, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX,  
  INT_MAX, INT_MAX, INT_MAX, INT_MAX], // Amount 0
```

```

[0, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**,
**10**, **11**], // Coin 1
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], // Coin 2
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] // Coin 5
]

```

Step 3: Consider Coin 2 (value = 2)

- For amounts $j < 2$, we cannot include Coin 2, so $R[2][j] = R[1][j]$.
- For amounts $j \geq 2$, we compare $R[1][j]$ and $1 + R[2][j - \text{coins}[1]]$.

```

R = [
[0, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX,
INT_MAX, INT_MAX, INT_MAX, INT_MAX], // Amount 0
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], // Coin 1
[0, 1, **1**, 2, **2**, 3, **3**, 4, **4**, 5, **5**, 6], // Coin 2
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] // Coin 5
]

```

Step 4: Consider Coin 5 (value = 5)

- For amounts $j < 5$, we cannot include Coin 5, so $R[3][j] = R[2][j]$.
- For amounts $j \geq 5$, we compare $R[2][j]$ and $1 + R[3][j - \text{coins}[2]]$.

```

R = [
[0, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX,
INT_MAX, INT_MAX, INT_MAX, INT_MAX], // Amount 0
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], // Coin 1
[0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6], // Coin 2
[0, 1, 1, 2, 2, **1**, 2, **2**, 3, **3**, 2, **3**] // Coin 5
]

```

In the final table, the highlighted values show the minimum number of coins needed for each amount using the available coins. The value $R[3][11] = 3$ is the minimum number of coins needed to make up the amount 11.

This visualization shows the decision-making process at each step, considering whether to include a coin based on its value, and updating the table accordingly. The highlighted values indicate the choices made by the algorithm at each step to achieve the minimum number of coins. If you need further clarification or have more questions, feel free to ask!

Longest common sub-sequence problem.

Certainly! Let's visualize the dynamic programming table R for the Longest Common Subsequence (LCS) problem step by step, highlighting the current values being considered and updated. We'll use an example with $A = \{a, b, c, d, e\}$ and $B = \{b, c, e, f, g\}$ and lengths $N = 5$ and $M = 5$.

Step 1: Initialization

```
R = [  
  [0, 0, 0, 0, 0, 0], // LCS with B[0...M] and A[0]  
  [0, 0, 0, 0, 0, 0], // LCS with B[0...M] and A[1]  
  [0, 0, 0, 0, 0, 0], // LCS with B[0...M] and A[2]  
  [0, 0, 0, 0, 0, 0], // LCS with B[0...M] and A[3]  
  [0, 0, 0, 0, 0, 0], // LCS with B[0...M] and A[4]  
  [0, 0, 0, 0, 0, 0]  // LCS with B[0...M] and A[5]  
]
```

Step 2: Computation

- We fill the table R row by row, column by column.
- For each cell $R[i][j]$, if $A[i] == B[j]$, we set $R[i][j]$ to $1 + R[i-1][j-1]$.
- If $A[i] != B[j]$, we set $R[i][j]$ to $\max(R[i-1][j], R[i][j-1])$.

Visualization:

```
A = [-, a, b, c, d, e]  
B = [-, b, c, e, f, g]
```

```
R = [  
  [0, 0, 0, 0, 0, 0], // LCS with B[0...M] and A[0]  
  [0, 0, 1, 1, 1, 1], // LCS with B[0...M] and A[1]  
  [0, 1, 1, 2, 2, 2], // LCS with B[0...M] and A[2]  
  [0, 1, 2, 2, 3, 3], // LCS with B[0...M] and A[3]  
  [0, 1, 2, 2, 3, 3], // LCS with B[0...M] and A[4]  
  [0, 1, 2, 2, 3, 3]  // LCS with B[0...M] and A[5]  
]
```

- The first row and column remain 0 after initialization because an LCS with an empty string is 0.
- When comparing $A[1]$ with B , we find a common subsequence of length 1 at $B[2]$ (both are b).
- When comparing $A[2]$ with B , we extend the common subsequence to length 2 at $B[3]$ (both are c).
- When comparing $A[3]$ with B , there's no match, so we carry over the maximum length from the previous computations.

- When comparing $A[4]$ with B , there's no match, so we carry over the maximum length from the previous computations.
- When comparing $A[5]$ with B , we find a common subsequence of length 3 at $B[3]$ (both are e).

The final result $R[N][M]$ or $R[5][5]$ is 3, which is the length of the longest common subsequence between A and B .

This visualization shows the decision-making process at each step, considering whether characters match and updating the table accordingly. The highlighted values indicate the choices made by the algorithm at each step to achieve the longest common subsequence. If you need further clarification or have more questions, feel free to ask

Rod Cutting Problem

Certainly! Let's dive into the **Rod Cutting** problem and understand how dynamic programming can help us find the optimal solution. I'll break it down step by step.

Rod Cutting Problem

Suppose you have a rod of length N , and you want to cut it into smaller pieces to maximize the total amount of money you can get. Each piece of length i is worth $P[i]$ dollars. Your goal is to find the best way to cut the rod to maximize the total value.

Approach

We'll use dynamic programming to solve this problem. The key idea is to build a table $R[0...N]$ where $R[i]$ represents the maximum value we can obtain from a rod of length i . We'll compute the values of $R[i]$ iteratively based on smaller subproblems.

1. **Initialization:** Set $R[0] = 0$ (since a rod of length 0 has no value).
2. **Computation:**
 - For each j from 1 to N , compute $R[j]$ as follows:
 - Initialize a variable q to 0.
 - For each i from 1 to j , calculate q as the maximum of:
 - The value of the piece of length i ($P[i]$) plus the maximum value obtained from the remaining rod of length $j - i$ ($R[j - i]$).
 - Set $R[j] = q$.
3. **Return:** The final answer is $R[N]$, which represents the maximum value achievable for the entire rod.

Step-by-Step Explanation and Visualization

Let's walk through an example with a rod of length $N = 5$ and given values $P[] = [0, 2, 5, 9, 10]$:

1. **Initialization:**
 - $R[0] = 0$
2. **Computation:**
 - For $j = 1$:
 - No cuts possible, so $R_1 = P_1 = 2$.
 - For $j = 2$:
 - Consider cutting at $i = 1$: $q = \max(0, P_1 + R[2 - 1]) = \max(0, 2 + 0) = 2$.
 - Set $R_2 = 2$.
 - For $j = 3$:
 - Consider cutting at $i = 1$: $q = \max(0, P_1 + R[3 - 1]) = \max(0, 2 + 2) = 4$.
 - Consider cutting at $i = 2$: $q = \max(4, P_2 + R[3 - 2]) = \max(4, 5 + 0) = 5$.
 - Set $R_3 = 5$.
 - For $j = 4$:
 - Consider cutting at $i = 1$: $q = \max(0, P_1 + R[4 - 1]) = \max(0, 2 + 5) = 7$.
 - Consider cutting at $i = 2$: $q = \max(7, P_2 + R[4 - 2]) = \max(7, 5 + 2) = 7$.
 - Consider cutting at $i = 3$: $q = \max(7, P_3 + R[4 - 3]) = \max(7, 9 + 0) = 9$.
 - Set $R[4] = 9$.
 - For $j = 5$:
 - Consider cutting at $i = 1$: $q = \max(0, P_1 + R[5 - 1]) = \max(0, 2 + 9) = 11$.
 - Consider cutting at $i = 2$: $q = \max(11, P_2 + R[5 - 2]) = \max(11, 5 + 9) = 14$.
 - Consider cutting at $i = 3$: $q = \max(14, P_3 + R[5 - 3]) = \max(14, 9 + 5) = 14$.
 - Consider cutting at $i = 4$: $q = \max(14, P[4] + R[5 - 4]) = \max(14, 10 + 0) = 14$.
 - Set $R[5] = 14$.
3. **Final Answer:** The maximum value for a rod of length 5 is $R[5] = 14$

