

# Exercise 4

## Explanation:

We use Dynamic Programming to build a 2D table K where:

- $K[i][w]$  represents the maximum profit achievable using the first  $i$  items with a weight capacity of  $w$ .

Steps:

- Initialization:
  - Create a table K of size  $(n+1) \times (\text{max\_weight\_per\_saddles}+1)$  initialized to 0.
  - Rows represent the number of items considered.
  - Columns represent weight capacities.
- Table Population:
  - For each item  $i$  and weight capacity  $w$ :
    - If the item's weight is less than or equal to  $w$ :
      - Decide whether to include or exclude the item:
        - Include: Add the item's profit to the remaining capacity's profit.
        - Exclude: Use the previous maximum profit.
      - Take the maximum of these two options.

## Code:

```
def knapsack(treasure_profit, treasure_weight, max_weight_per_saddles):
    n = len(treasure_profit)
    # Create a 2D array to store the maximum value at each i (number of items) and w (max weight)
    # Initialize the K array with 0s
    #  $K[i][w]$  will hold the maximum value of the first  $i$  items with a maximum weight  $w$ 
    K = [[0 for _ in range(max_weight_per_saddles + 1)] for _ in range(n + 1)]
    # Build the K array
    for i in range(1, n + 1): # Iterate through each item
        for w in range(max_weight_per_saddles + 1): # Iterate through each weight
            # If the weight of the current item is less than or equal to the current weight w
            # we have two options: include the item or not
            if treasure_weight[i - 1] <= w: # We check if the current item could be included in the knapsack if it were empty
                # We take the maximum of including the item or not including it
                #  $K[i - 1][w]$  is the max value without including the current item, so the previous value
                #  $K[i - 1][w - \text{treasure\_weight}[i - 1]] + \text{treasure\_profit}[i - 1]$  is the max value including the current item
                K[i][w] = max(K[i - 1][w], K[i - 1][w - treasure_weight[i - 1]] + treasure_profit[i - 1])
            else: # If the weight of the current item is greater than the current weight w, we cannot include it
                K[i][w] = K[i - 1][w]

    return K[n][max_weight_per_saddles], K

# Example data
treasure_profit = [1,6,4,6,2,4,6,7,4,8,9,5,3,2,1,5,4,3,2,1]
treasure_weight = [2,5,7,4,3,1,4,5,7,9,6,2,5,7,9,5,3,2,5,7]
max_weight_saddles = 10

# Call the knapsack function
max_profit, K = knapsack(treasure_profit, treasure_weight, max_weight_saddles)
print("Maximum profit:", max_profit)

# Print the matrix K in a nice format
```

```
print("K matrix:")
for i in range(len(K)):
    for j in range(len(K[i])):
        print(f"{K[i][j]:3}", end=" ")
    print()
```

## Test case:

```
treasure_profit = [1,6,4,6,2,4,6,7,4,8,9,5,3,2,1,5,4,3,2,1]
treasure_weight = [2,5,7,4,3,1,4,5,7,9,6,2,5,7,9,5,3,2,5,7]
```

```
max_weight_saddles = 10
```

```
Maximum profit: 19
```

```
K matrix:
```

```
0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1
0 0 1 1 1 6 6 7 7 7 7 7
0 0 1 1 1 6 6 7 7 7 7 7
0 0 1 1 6 6 7 7 7 12 12
0 0 1 2 6 6 7 8 8 12 12
0 4 4 5 6 10 10 11 12 12 16
0 4 4 5 6 10 10 11 12 16 16
0 4 4 5 6 10 11 11 12 16 17
0 4 4 5 6 10 11 11 12 16 17
0 4 4 5 6 10 11 11 12 16 17
0 4 4 5 6 10 11 13 13 16 17
0 4 5 9 9 10 11 15 16 18 18
0 4 5 9 9 10 11 15 16 18 18
0 4 5 9 9 10 11 15 16 18 18
0 4 5 9 9 10 11 15 16 18 18
0 4 5 9 9 10 11 15 16 18 18
0 4 5 9 9 10 13 15 16 18 19
0 4 5 9 9 12 13 15 16 18 19
0 4 5 9 9 12 13 15 16 18 19
0 4 5 9 9 12 13 15 16 18 19
```

## Exercise 7

### Explanation:

The code solves the Longest Common Subsequence (LCS) problem using dynamic programming. It builds a 2D table L where  $L[i][j]$  stores the length of the LCS for the first  $i$  elements of A and the first  $j$  elements of B. If  $A[i-1] == B[j-1]$ , the value is incremented from the diagonal ( $L[i-1][j-1] + 1$ ); otherwise, it takes the maximum of the values from the previous row or column. After filling the table, the LCS length is found at  $L[m][n]$ . The LCS itself is reconstructed by backtracking through the table, appending matching elements and moving diagonally, or moving up or left based on the larger value. The result is reversed to get the correct order.

### Code:

```
def find_max_shared_substring(A, B):
```

```

# Create a 2D array to store the lengths of longest common subsequences
m = len(A)
n = len(B)
L = [[0] * (n + 1) for _ in range(m + 1)]

# Build the L array
for i in range(1, m + 1): # Iterate through each element of A
    for j in range(1, n + 1): # Iterate through each element of B
        # If the current elements of A and B match, increment the length of the common subsequence
        if A[i - 1] == B[j - 1]:
            L[i][j] = L[i - 1][j - 1] + 1 # Increment the length of the common subsequence
        # If they do not match, take the maximum length from the previous elements
        else:
            L[i][j] = max(L[i - 1][j], L[i][j - 1]) # The maximum length is the maximum of the two previous lengths
            (previous row or previous column)

# Length of the longest common subsequence
length = L[m][n]

# Backtrack to find the common subsequence
i, j = m, n
common_subsequence = []
while i > 0 and j > 0: # Backtrack through the L array
    if A[i - 1] == B[j - 1]: # If the current elements of A and B match, it is part of the common subsequence
        common_subsequence.append(A[i - 1])
        i -= 1
        j -= 1
    elif L[i - 1][j] > L[i][j - 1]: # If the value above is greater, move up
        i -= 1
    else: # If the value to the left is greater, move left
        j -= 1

common_subsequence.reverse() # Reverse to get the correct order
return length, common_subsequence

```

```

A = [0,1,1,0,1,0,1,0]
B = [1,0,1,0,0,1,0,0,1]

```

```

l, s = find_max_shared_substring(A, B)
print("Length of longest common subsequence:", l)
print("Longest common subsequence:", s)

```

## Test case:

```

A = [0,1,1,0,1,0,1,0]
B = [1,0,1,0,0,1,0,0,1]

```

Length of longest common subsequence: 6  
Longest common subsequence: [1, 0, 1, 0, 1, 0]