Hands-on Activity 2.1 : Dynamic Programming

Objective(s):

This activity aims to demonstrate how to use dynamic programming to solve problems.

Intended Learning Outcomes (ILOs):

- Differentiate recursion method from dynamic programming to solve problems.
- Demonstrate how to solve real-world problems using dynamic programming

Resources:

- Jupyter Notebook

Procedures:

1. Create a code that demonstrate how to use recursion method to solve problem

2. Create a program codes that demonstrate how to use dynamic programming to solve the same problem

Question:

Explain the difference of using the recursion from dynamic programming using the given sample codes to solve the same problem

Type your answer here:

3. Create a sample program codes to simulate bottom-up dynamic programming

4. Create a sample program codes that simulate tops-down dynamic programming

Question:

Explain the difference between bottom-up from top-down dynamic programming using the given sample codes

Although they take different approaches, all three codes address the knapsack problem. Recursion is simple yet slow for complex situations because it involves several calculations. Dynamic programming is more effective since it saves solutions to subproblems and eliminates duplication. Last but not least, memorization is top-down and comparable to dynamic programming; it might be superior in some circumstances. To summarize, the recursive technique is optimized via dynamic programming and memoization, which reduces complexity in terms of time and space.Type your answer here:

0/1 Knapsack Problem

- Analyze three different techniques to solve knapsacks problem

1. Recursion
2. Dynamic Programming
3. Memoization

```
#sample code for knapsack problem using recursion
def rec_knapSack(w, wt, val, n):

  #base case
  #defined as nth item is empty;
  #or the capacity w is 0
  if n == 0 or w == 0:
    return 0

  #if weight of the nth item is more than
  #the capacity W, then this item cannot be included
  #as part of the optimal solution
  if(wt[n-1] > w):
    return rec_knapSack(w, wt, val, n-1)

  #return the maximum of the two cases:
  # (1) include the nth item
  # (2) don't include the nth item
  else:
    return max(
        val[n-1] + rec_knapSack(
            w-wt[n-1], wt, val, n-1),
```

```
            rec_knapSack(w, wt, val, n-1)
    )

#To test:
val = [60, 100, 120] #values for the items
wt = [10, 20, 30] #weight of the items
w = 50 #knapsack weight capacity
n = len(val) #number of items

rec_knapSack(w, wt, val, n)
```

```
220
```

```
#Dynamic Programming for the Knapsack Problem
def DP_knapSack(w, wt, val, n):
  #create the table
  table = [[0 for x in range(w+1)] for x in range (n+1)]

  #populate the table in a bottom-up approach
  for i in range(n+1):
    for w in range(w+1):
      if i == 0 or w == 0:
        table[i][w] = 0
      elif wt[i-1] <= w:
        table[i][w] = max(val[i-1] + table[i-1][w-wt[i-1]],
                          table[i-1][w])
  return table[n][w]

#To test:
val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
n = len(val)

DP_knapSack(w, wt, val, n)
```

```
220
```

```
#Sample for top-down DP approach (memoization)
#initialize the list of items
val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
n = len(val)

#initialize the container for the values that have to be stored
#values are initialized to -1
calc =[[-1 for i in range(w+1)] for j in range(n+1)]
```

```
def mem_knapSack(wt, val, w, n):
  #base conditions
  if n == 0 or w == 0:
    return 0
  if calc[n][w] != -1:
    return calc[n][w]

  #compute for the other cases
  if wt[n-1] <= w:
    calc[n][w] = max(val[n-1] + mem_knapSack(wt, val, w-wt[n-1], n-1),
                     mem_knapSack(wt, val, w, n-1))
    return calc[n][w]
  elif wt[n-1] > w:
    calc[n][w] = mem_knapSack(wt, val, w, n-1)
    return calc[n][w]

mem_knapSack(wt, val, w, n)
```

```
220
```

Code Analysis Recursive decision logic is easily comprehensible, straightforward, and readable. Nevertheless, it's not that much. Calculations that are repeated make it inefficient. Due to its readability, recursion may be chosen for straightforward issues or small datasets.

In comparison to recursion, the dynamic version of the code is slightly less clear and needs the creation and maintenance of a 2D table. However, it is more efficient, avoids unnecessary calculations, and builds the ideal solution via a bottom-up method. Dynamic programming provides significant efficiency advantages for larger datasets or performance problems.

The memorization version utilizes a top-down approach and avoids pointless calculations like recursion; it may be superior for some issue structures. Though it may not always beat DP in general situations, memorization might be a strong option for problems with particular structures where top-down logic is advantageous.

Seatwork 2.1

Task 1: Modify the three techniques to include additional criterion in the knapsack problems

```
#type your code here
#Recursion
```

```
#Dynamic
```

```
#Memoization

###Recursion###
def rec_knapSack(w, wt, val, age, n):

    # Base cases
    if n == 0 or w == 0:
        return 0

    # If the weight of the nth item is more than the capacity w, exclude it
    if wt[n-1] > w:
        return rec_knapSack(w, wt, val, age, n-1)

    # Check if the age of the food item exceeds the threshold
    if age[n-1] > MAX_AGE:
        return rec_knapSack(w, wt, val, age, n-1)

    else:
        return max(
            val[n-1] + rec_knapSack(w - wt[n-1], wt, val, age, n-1),
            rec_knapSack(w, wt, val, age, n-1)
        )


wt = [1, 3, 4, 5] #weight value
val = [10, 40, 50, 70] #item value
age = [2, 1, 4, 3] #food item age
MAX_AGE = 3  # Maximum allowed age
w = 7  # Knapsack capacity

# Calculate the maximum value
max_value = rec_knapSack(w, wt, val, age, len(val))

# Print the result
print("(Recursion) Maximum value considering the food item age and weight capacity:", max
```

```
(Recursion) Maximum value considering the food item age and weight capacity: 80
```

```
###Dynamic###
def DP_knapSack(w, wt, val, age, n, MAX_AGE):

    # Create the table with an extra dimension for age
    table = [[[0 for _ in range(MAX_AGE + 1)] for _ in range(w + 1)] for _ in range(n + 1

    # Populate the table in a bottom-up approach
```

```
    for i in range(n + 1):
        for w in range(w + 1):
            for a in range(MAX_AGE + 1):
                if i == 0 or w == 0 or a == 0:
                    table[i][w][a] = 0
                elif wt[i - 1] <= w and age[i - 1] <= a:
                    table[i][w][a] = max(
                        val[i - 1] + table[i - 1][w - wt[i - 1]][a - 1],
                        table[i - 1][w][a]
                    )
                else:
                    table[i][w][a] = table[i - 1][w][a]

    return table[n][w][MAX_AGE]  # Maximum value considering age and weight



wt = [1, 3, 4, 5] #weight value
val = [10, 40, 50, 70] #item value
age = [2, 1, 4, 3] #food item age
MAX_AGE = 3  # Maximum allowed age
w = 7  # Knapsack capacity

# Calculate the maximum value
max_value = rec_knapSack(w, wt, val, age, len(val))

# Print the result
print("(Dynamic Programming) Maximum value considering the food item age and weight capac
```

```
 (Dynamic Programming) Maximum value considering the food item age and weight capacity
```

```
###Memoization###

def mem_knapSack(wt, val, age, w, n, MAX_AGE):

    # Initialize the memoization table with an extra dimension for age
    calc = [[[-1 for _ in range(MAX_AGE + 1)] for _ in range(w + 1)] for _ in range(n + 1)]

    def _mem_knapSack(i, w, a):
        # Base cases
        if i == 0 or w == 0 or a == 0:
            return 0
        if calc[i][w][a] != -1:
            return calc[i][w][a]

        # Include item if weight and age constraints are met
        if wt[i - 1] <= w and age[i - 1] <= a:
            calc[i][w][a] = max(
                val[i - 1] + _mem_knapSack(i - 1, w - wt[i - 1], a - 1),
                _mem_knapSack(i - 1, w, a)
            `
```

```
            )
            return calc[i][w][a]

        # Exclude item
        else:
            calc[i][w][a] = _mem_knapSack(i - 1, w, a)
            return calc[i][w][a]

    return _mem_knapSack(n, w, MAX_AGE)

# Sample data (including ages)
val = [10, 40, 50, 70]
wt = [1, 3, 4, 5]
age = [2, 1, 4, 3]
w = 7
n = len(val)
MAX_AGE = 3  # Maximum allowed age

max_value = mem_knapSack(wt, val, age, w, n, MAX_AGE)
print("(Memoization) Maximum value considering age and weight capacity:", max_value)
```

```
(Memoization) Maximum value considering age and weight capacity: 80
```

Fibonacci Numbers

Task 2: Create a sample program that find the nth number of Fibonacci Series using Dynamic Programming

```
def Dyn_Fib(n):
    if n < 0:
        raise ValueError("n must be non-negative")

    # Calculate Fibonacci numbers iteratively, storing them in a list
    dp = [0] * (n + 2)
    dp[0], dp[1] = 0, 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]

def main():
    while True:
        try:
            n = int(input("Enter a non-negative integer to find its Fibonacci number: "))
            if n >= 0:
                break
```

```
            else:
                print("n must be non-negative. Please try again.")
        except ValueError:
            print("Invalid input. Please enter a valid integer.")

    fib_num = Dyn_Fib(n)
    print(f"The {n}th Fibonacci number is {fib_num}")

if __name__ == "__main__":
    main()
```

```
Enter a non-negative integer to find its Fibonacci number: 10
The 10th Fibonacci number is 55
```

Supplementary Problem (HOA 2.1 Submission):

- Choose a real-life problem
- Use recursion and dynamic programming to solve the problem

After a war a caravan of refugees would like to go on a long journey for a better life in another kingdom, however the caravan leaders noticed that the caravan members brought variety of unbalanced weighted items and most of the are tired. How should the caravan elader distribute the item weight.How should the caravan leader distribute the weight of the cargo to all the members with a specific BMI value.

```
def dynamic_knapSack(weights, bmis, ages, capacity, num_items, max_age):
  dp = [[[0 for _ in range(max_age + 1)] for _ in range(capacity + 1)] for _ in range(num

  for i in range(1, num_items + 1):
    for j in range(1, capacity + 1):
      for a in range(1, max_age + 1):
        # If the item weight and age are within limits, consider including it.
        if weights[i - 1] <= j and ages[i - 1] <= a:
          # Choose the maximum BMI achievable
          dp[i][j][a] = max(
              bmis[i - 1] + dp[i - 1][j - weights[i - 1]][a - 1],  # Include item
              dp[i - 1][j][a],  # Exclude item
          )
        else:
          # exclude it.
          dp[i][j][a] = dp[i - 1][j][a]

  # Return the recommended weight
  return dp[num_items][capacity][max_age]
```

```
bmi = [20, 30, 21, 19]  # BMI values
wt = [5, 10, 20, 6]  # Weight values
age = [19, 25, 40, 19]  # Age of the person
w = 20  # Maximum weight capacity per person
n = len(bmi)
MAX_AGE = 50  # Maximum allowed age

max_value = dynamic_knapSack(wt, bmi, age, w, n, MAX_AGE)
print("(Dynamic Programming) Recommended cargo distribution BMI assignability:", max_valu
```

```
(Dynamic Programming) Recommended cargo distribution BMI assignability: 50
```

```
def recursion_knapSack(weights, bmis, ages, capacity, num_items, max_age):
  if num_items == 0 or capacity == 0 or max_age == 0:
    return 0

  if weights[0] <= capacity and ages[0] <= max_age:
    # Choose including or excluding the item.
    include = bmis[0] + dynamic_knapSack(weights[1:], bmis[1:], ages[1:], capacity - weig
    exclude = dynamic_knapSack(weights[1:], bmis[1:], ages[1:], capacity, num_items - 1,
    return max(include, exclude)
  else:
    #exclude it.
    return dynamic_knapSack(weights[1:], bmis[1:], ages[1:], capacity, num_items - 1, max

bmi = [20, 30, 21, 19]  # BMI values
wt = [5, 10, 20, 6]  # Weight value
age = [19, 25, 40, 19]  # Age of the person
w = 20  # Maximum weight capacity per person
n = len(bmi)
MAX_AGE = 50  # Maximum allowed age

max_value = dynamic_knapSack(wt, bmi, age, w, n, MAX_AGE)
print("(Recursion Programming) Recommended cargo distribution BMI assignability:", max_valu
```

```
(Recursion Programming) Recommended cargo distribution BMI assignability: 50
```

Conclusion

This Activity explored three approaches to the knapsack problem: clear but inefficient recursion, efficient but table-driven dynamic programming (DP), and top-down, calculation-saving memorization. For most knapsack dilemmas, DP reigns supreme. Its bottom-up logic and efficiency make it ideal for larger datasets and performance-focused situations. While DP might lack recursion's immediate grasp, its clear structure and power outweigh that initial learning

curve. Memorization emerges as a contender for specific problem structures where top-down logic thrives. If DP doesn't offer substantial efficiency gains, memorization's targeted calculation avoidance could shine. However, it's not a universal champion, and DP remains the default choice. Leave recursion behind for educational purposes or tiny datasets. Its repeated calculations make it impractical for real-world scenarios. Lastly, the optimal approach hinges on your unique problem characteristics. Prioritize readability, efficiency, and problem structure for a tailored solution. Don't hesitate to experiment and find the knapsack conqueror that fits your needs!