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

```
def rec_add_numbers(x):
    """Get the sum of all numbers from declared number to 1 using
recursive method"""
    if x == 0 or x == 1:
        return 1
    else:
        return x + rec_add_numbers(x-1)

num = 8
print(rec_add_numbers(num))
```

```
36
```

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

```
def dyn_add_numbers(x):
    """Get the sum of all numbers from declared number to 1 using
dynamic programming"""
    memo = {0: 0}
    for i in range(1, x+1):
        memo[i] = memo[i-1] + i
    return memo[x]

num = 8
print(dyn_add_numbers(num))
```

```
36
```

Question:

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

- In the recursion method, the program runs the given function repeatedly, leading to run more steps to get to the answer, and as for the dynamic programming method, it has the ability to store the values used recently and it adds these values altogether in the end. In this situation, the dynamic programming performs better, easier to understand, and requires less code with its memoization feature.

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

```
def bu_piggy_bank(days, deposit):
    """Calculate the total amount of money in piggy bank after certain
number of days
    with fixed daily deposit using bottom-up dynamic programming"""
    dp = [0] * (days+1)
    for n in range(1, days+1):
        dp[n] = dp[n-1] + deposit
    return dp

result = bu_piggy_bank(10, 50)
print(result[10])
```

```
500
```

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

```
def td_piggy_bank(days, deposit):
    """Calculate the total amount of money in piggy bank after certain
number of days
    with fixed daily deposit using top-down dynamic programming"""
    memo = {0: 0}
    for n in range(1, days+1):
        memo[n] = memo[n-1] + deposit
    return memo

result = td_piggy_bank(10, 50)
print(result[10])
```

```
500
```

Question:

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

- Using the top-down dynamic programming approach ensures efficiency in every computation made. However, this method runs the function multiple times since it needs to iterate through each day to get the total savings. This is where the bottom-up dynamic programming approach becomes a better option in this scenario. This method ensures a faster performance and less code since it does not use recursion to get the total savings.

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

1. The recursion method involves repeatedly running the process of evaluating if a specific item can be added to the knapsack and computing for the total value that the knapsack can carry. In terms of performance, it is considered a slow approach, however, in terms of efficiency, it can be considered a great approach for its accurate results.
2. The bottom-up dynamic programming method involves iterating through the items, storing the ones that can fit in the knapsack in a table, and returning the overall value of the items in this table. In terms of its usability, it has a better performance than the recursion method as this method eliminates the use of recursion, and a good accuracy. In short, this method is more on iterating through the items and appending them in the knapsack.
3. The top-down dynamic programming approach involves running the function multiple times to get the items that fit the knapsack and their overall value. Its performance is better than the recursive method and it has great accuracy. However, it is worse than the bottom up approach, requires more code, and harder to understand.

Seatwork 2.1

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

```
# items
val = [60, 100, 120] #values for the items
wt = [10, 20, 30] #weight of the items
vol = [5, 10, 20] # volume of the items

# criteria
w = 50 #knapsack weight capacity
v = 25
n = len(val) #number of items

# Recursion

def rec_knapSack(w, wt, v, vol, val, n):
  if n == 0 or w == 0 or v == 0:
    return 0

  if (wt[n-1] > w) or (vol[n-1] > v):
    return rec_knapSack(w, wt, v, vol, val, n-1)

  else:
    include = val[n-1] + rec_knapSack(w - wt[n-1], wt, v - vol[n-1],
vol, val, n-1)
    exclude = rec_knapSack(w, wt, v, vol, val, n-1)
    return max(include, exclude)

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

```
180
```

```
# Dynamic
```

```
def DP_knapSack(w, wt, v, vol, val, n):
    table = [[[0 for x in range(v+1)] for x in range(w+1)] for x in
range(n+1)]
    for i in range(n+1):
        for w in range(w+1):
            for v in range(v+1):
                if i == 0 or w == 0 or v == 0:
                    table[i][w][v] = 0
                elif wt[i-1] <= w and vol[i-1] <= v:
                    include = val[i-1] + table[i-1][w - wt[i-1]][v -
vol[i-1]]
                    exclude = table[i-1][w][v]
                    table[i][w][v] = max(include, exclude)
                else:
                    table[i][w][v] = table[i-1][w][v]
    return table[n][w][v]

print(DP_knapSack(w, wt, v, vol, val, n))
```

```
180
```

```
# Memoization

calc =[[[-1 for i in range(w+1)] for j in range(w+1)] for k in
range(n+1)]

def mem_knapSack(w, wt, v, vol, val, n):
  if n == 0 or w == 0 or v == 0:
    return 0
  if calc[n][w][v] != -1:
    return calc[n][w][v]

  if wt[n-1] <= w or vol[n-1] <= v:
    include = val[n-1] + mem_knapSack(w-wt[n-1], wt, v-vol[n-1], v,
val, n-1)
    exclude = mem_knapSack(w, wt, vol, v, val, n-1)
    calc[n][w][v] = max(include, exclude)
    return calc[n][w][v]
  elif wt[n-1] > w or vol[n-1] > v:
    calc[n][w][v] = mem_knapSack(w, wt, vol, v, val, n-1)
    return calc[n][w][v]

print(DP_knapSack(w, wt, v, vol, val, n))
```

```
180
```

Fibonacci Numbers

```
def rec_fib_seq(n):
    if n <= 1:
```

```
        return n
    else:
        return(rec_fib_seq(n-2) + rec_fib_seq(n-1))

num = 8
print(rec_fib_seq(num))
```

```
21
```

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

```
#type your code here
def dyn_fib_seq(n):
  Memo = {}
  def fib(n):
    if n in Memo:
      return Memo[n]
    if n == 1:
      return 0
    if n == 2:
      return 1
    else:
      x = fib(n-2) + fib(n-1)
      Memo[n] = x
      return x
  return fib(n)

num = 8
print(dyn_fib_seq(num))
```

```
13
```

Supplementary Problem (HOA 2.1 Submission):

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

```
#type your code here for recursion programming solution
def rec_savings(years, deposit, rate):
    if years == 0:
        return 0
    else:
        return  rec_savings(years-1, deposit, rate) * (1 + rate) +
deposit

result = rec_savings(10, 1000, 0.05)
print(result)
```

```
12577.892535548832
```

```
#type your code here for dynamic programming solution
def dyn_savings(years, deposit, rate):
    memo = {0: 0}
    for n in range(1, years+1):
        memo[n] = memo[n-1] * (1 + rate) + deposit
    return memo

result = dyn_savings(10, 1000, 0.05)
print(result[10])
```

```
12577.892535548832
```

Conclusion

- This activity mainly shows the difference between different programming methods discussed in the previous lectures: the recursive method, bottom-up dynamic programming method, and the top-down dynamic programming method by observing their performances in solving different scenarios. The recursive method is best for efficiency and accuracy of final results by starting from the base case and runs the same function recursively to solve the sub problems. The bottom-up dynamic programming method is a bit better approach than the recursive method. It still uses the recursion concept but it adds a memorization feature in order to store the solutions from recent sub problems. The bottom-up dynamic programming method is the best approach among the three. It eliminates the use of recursion and instead uses iteration to solve the sub problems and store their solutions in a table to get the final result. To conclude, the mentioned programming methods possess their own function, strengths, and weaknesses. It is important to know them so we can use them properly in solving different problems.