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

```
# Function to implement Fibonacci Series
def Fibonacci(n):
    tb = [0, 1]
    for i in range(2, n+1):
      tb.append(tb[i-1] + tb[i-2])
    return tb

print('The Fibonacci of 10 is:',Fibonacci(10))
```

```
 The Fibonacci of 10 is: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

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

```
#The top-down is a recursive problem solving approach, the sample code below
#shows a recursive way to solve a factorial.
def Factorial(n):
  if n == 0 or n == 1:
    return 1

  else:
    return(n * Factorial(n-1))

n = 10
print('The factorial of', n,'is:',Factorial(n))
```

```
 The factorial of 10 is: 3628800
```

Question:

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

Type your answer here:

The bottom-up method uses iteratively solutions to solve the problem from smallest to largest. While on the other hand, the top-down solves the problem recursively and can use the technique of memoization.

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

Recursion

The recursion technique checks first the value of the weight and the number of items, if the number of items is more than the weight capacity, then the value will not be included as said.

Dynamic Programming

For the Dynamic technique, It first creates a table then populates the table with the bottom-up approach. The table is being populated by implementing a iteration method.

Memoization

Lastly for the Memoization, as we can observe in the sample code, we can notice that the Code is implemented with a top-down approach. It first starts by declaring values in a list, then values are initialized to -1. After that, it now checks if the number of items or the weight capacity is equals to 0. if the number of items and the weight capacity is not equals to -1, it will then just return the value. After creating the conditions, the code then implements a computation for the other cases of the values.

Seatwork 2.1

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

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

  if n == 0 or w == 0:
    return 0

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

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

#In this criterion I added a new criterion which is the price of the items.
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
p = [70, 85, 90] #price of the items

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

```
220
```

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

  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]

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

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

```
220
```

Fibonacci Numbers

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

```
#type your code here
"Already submitted last Activity."
```

Supplementary Problem (HOA 2.1 Submission):

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

```
#A dad wants his young son to learn simple math operations.
#With this simple python code, it can calculate simple math operations
#And it will show the output of the users inputs.

def calculate():
    # Get user input for the operation
    operation = input("Enter operation (+, -, *, /) or 'exit' to end: ")

    if operation == 'exit':
        return 0
```

```
    #user inputs
    n1 = float(input("Enter the first number: "))
    n2 = float(input("Enter the second number: "))

    #operation
    result = 0
    if operation == '+':
        result = n1 + n2
    elif operation == '-':
        result = n1 - n2
    elif operation == '*':
        result = n1 * n2
    elif operation == '/':

        if n2 == 0:
            print("Error: Cannot divide by zero.")
            return calculate()
        result = n1 / n2
    else:
        print("Invalid operation. Please enter +, -, *, or /.")
        return calculate()

    print(f"Result: {result}")
    return calculate()

calculate()
```

```
 Enter operation (+, -, *, /) or 'exit' to end: +
 Enter the first number: 10
 Enter the second number: 20
 Result: 30.0
 Enter operation (+, -, *, /) or 'exit' to end: -
 Enter the first number: 5.5
 Enter the second number: 2.5
 Result: 3.0
 Enter operation (+, -, *, /) or 'exit' to end: /
 Enter the first number: 10
 Enter the second number: 2
 Result: 5.0
 Enter operation (+, -, *, /) or 'exit' to end: exit
 0
```

Conclusion

In conclusion the recursion is very useful since with the help of recursion, specific problems can be optimized since recursion has a technique which a function can call itself. While on the other hand, the Dynamic Programming is also useful since it solves problems by combining the solutions with sub-problems.