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

0/1 Knapsack Problem

- Analyze three different techniques to solve knapsacks problem

1. Recursion
2. Dynamic Programming
3. Memoization

```
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
```

```
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

For the code that implements that knapsack problem without Dynamic Programming, the 'rec_Knapsack' takes the weight capacity, the weight and values of items, and the number of items as parameters. The base case of recursion has a conditional statement, as well as including and excluding items.

For the solution with the dynamic programming implementation, it has the same parameters wherein a table filled with the nested loop from the latter part of the code. Both are tested with sample values at the end, with the Dynamic Programming implementation using a Bottom-Up Approach.

For the Top-Down Approach, 'calc' is introduced as a table with memoization implementation following conditional statements and a memoization check later.

Seatwork 2.1

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

```
def rec_knapSack(w, wt, val, d, n):  # d is the additional criterion for the problem, den
  # Everything remains the same but with the additional criterion implememented.
  if n == 0 or w == 0:
```

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

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

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

val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
d = [] # d as the density criterion for the objects involved in the knapsack.
n = len(val)

for i in range(len(wt)):
  d.append(val[i]/wt[i])

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

```
11.0
```

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

```
def fbnci(n):
    # Conditional base cases
    if n <= 0:
        return 0
    elif n == 1:
        return 1

    fbnci_array = [0] * (n + 1)
    fbnci_array[0] = 0
    fbnci_array[1] = 1

    # Implementing a bottom-up approach
    for i in range(2, n + 1):
        fbnci_array[i] = fbnci_array[i - 1] + fbnci_array[i - 2]
    return fbnci_array[n]

# n is the sequential number in the Fibonacci Series that will be determined.
n = 12
print(f"In the Fibonacci Sequence, the value of term {n} is equal to {fbnci(12)}")
```

```
In the Fibonacci Sequence, the value of term 12 is equal to 144
```

Supplementary Problem (HOA 2.1 Submission):

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

The Chosen Problem:

An electrician wants to provide more utility in the house by adding more appliances designed to improve living conditions. When purchasing electronics, he wants to maximize the operating efficiency with the most number of utility appliances -- considering future savings by their energy rating units, along with the selling price.

```
#Programming Solution with both Dynamic Programming and Recursion Implementation
class Gadgets:
    def __init__(self, n, p, b):
        self.name = n
        self.price = p
        self.budget = b

    def getPrice(self):
        return self.price

    def getBudget(self):
        return self.budget

    def Savings(self):
        return self.getPrice() / self.getBudget()

    def __str__(self):
        return self.name + ': (Base Price: ' + str(self.price) + ', Energy Rating: ' + st

def Checklist(names, price, budget):
    menu = []
    for i in range(len(price)):
        menu.append(Gadgets(names[i], price[i], budget[i]))
    return menu

def greedy(items, maxCost, keyFunction):
    itemsCopy = sorted(items, key=keyFunction, reverse=True)
    result = []
    totalValue, totalCost = 0.0, 0.0
    for i in range(len(itemsCopy)):
        if (totalCost + itemsCopy[i].getBudget()) <= maxCost:
            result.append(itemsCopy[i])
            totalCost += itemsCopy[i].getBudget()
            totalValue += itemsCopy[i].getPrice()
    return (result, totalValue)

def testGreedy(items, constraint, keyFunction):
```

```
    taken, val = greedy(items, constraint, keyFunction)
    print('Total price of items taken =', val)
    for item in taken:
        print('->', item)

def testGreedys(gadget, maxUnits):
    print('Use Greedy Algorithm for the max energy unit savings of', maxUnits)
    testGreedy(gadget, maxUnits, Gadgets.getPrice)
    print('\nUse Greedy Algorithm for the max energy unit savings of', maxUnits)
    testGreedy(gadget, maxUnits, lambda x: 1 / x.getBudget())
    print('\nUse Greedy Algorithm for the max energy unit savings of', maxUnits)
    testGreedy(gadget, maxUnits, Gadgets.Savings)

def maxValMemo(toConsider, avail, memo=None): #Implements Memoization into the code, stor
    if memo is None:                          #replaces the brute force implementation. C
        memo = {}                             #loop given the constraints, using memo dic
    if (len(toConsider), avail) in memo:
        return memo[(len(toConsider), avail)]
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getBudget() > avail:
        result = maxValMemo(toConsider[1:], avail, memo)
    else:
        nextItem = toConsider[0]
        withVal, withToTake = maxValMemo(toConsider[1:], avail - nextItem.getBudget(), me
        withVal += nextItem.getPrice()
        withoutVal, withoutToTake = maxValMemo(toConsider[1:], avail, memo)
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    memo[(len(toConsider), avail)] = result
    return result

def testMaxVal(gadget, maxUnits, printItems=True):
    print('\nUsing Dynamic Programming for the energy unit savings of', maxUnits)
    val, taken = maxValMemo(gadget, maxUnits)
    print('Total cost of all gadgets =', val)
    if printItems:
        for item in taken:
            print('->', item)

names = ['Phone', 'Computer', 'Television', 'Heater', 'Airconditioner', 'Refrigerator', '
price = [7500, 27500, 3250, 31500, 22000, 600, 2450, 1600]
budget = [10, 7.5, 4.5, 7.5, 9, 8, 6, 9.5]
gadgetlist = Checklist(names, price, budget)

testGreedys(gadgetlist, 40)  # Testing with Greedy Algorithm of 40 energy units (Wh)
testMaxVal(gadgetlist, 20)   # Testing with Dynamic Programming of 20 energy units (Wh)
```

```
Use Greedy Algorithm for the max energy unit savings of 80
```

```
Total price of items taken = 9640.0
-> Heater: (Base Price: 3150, Energy Rating: 7.5Wh)
-> Computer: (Base Price: 2750, Energy Rating: 7.5Wh)
-> Airconditioner: (Base Price: 2200, Energy Rating: 9Wh)
-> Phone: (Base Price: 750, Energy Rating: 10Wh)
-> Television: (Base Price: 325, Energy Rating: 4.5Wh)
-> Microwave: (Base Price: 245, Energy Rating: 6Wh)
-> Electric Fan: (Base Price: 160, Energy Rating: 9.5Wh)
-> Refrigerator: (Base Price: 60, Energy Rating: 8Wh)

Use Greedy Algorithm for the max energy unit savings of 80
Total price of items taken = 9640.0
-> Television: (Base Price: 325, Energy Rating: 4.5Wh)
-> Microwave: (Base Price: 245, Energy Rating: 6Wh)
-> Computer: (Base Price: 2750, Energy Rating: 7.5Wh)
-> Heater: (Base Price: 3150, Energy Rating: 7.5Wh)
-> Refrigerator: (Base Price: 60, Energy Rating: 8Wh)
-> Airconditioner: (Base Price: 2200, Energy Rating: 9Wh)
-> Electric Fan: (Base Price: 160, Energy Rating: 9.5Wh)
-> Phone: (Base Price: 750, Energy Rating: 10Wh)

Use Greedy Algorithm for the max energy unit savings of 80
Total price of items taken = 9640.0
-> Heater: (Base Price: 3150, Energy Rating: 7.5Wh)
-> Computer: (Base Price: 2750, Energy Rating: 7.5Wh)
-> Airconditioner: (Base Price: 2200, Energy Rating: 9Wh)
-> Phone: (Base Price: 750, Energy Rating: 10Wh)
-> Television: (Base Price: 325, Energy Rating: 4.5Wh)
-> Microwave: (Base Price: 245, Energy Rating: 6Wh)
-> Electric Fan: (Base Price: 160, Energy Rating: 9.5Wh)
-> Refrigerator: (Base Price: 60, Energy Rating: 8Wh)

Using Dynamic Programming for the energy unit savings of 60
Total cost of all gadgets = 9580
-> Electric Fan: (Base Price: 160, Energy Rating: 9.5Wh)
-> Microwave: (Base Price: 245, Energy Rating: 6Wh)
-> Airconditioner: (Base Price: 2200, Energy Rating: 9Wh)
-> Heater: (Base Price: 3150, Energy Rating: 7.5Wh)
-> Television: (Base Price: 325, Energy Rating: 4.5Wh)
-> Computer: (Base Price: 2750, Energy Rating: 7.5Wh)
-> Phone: (Base Price: 750, Energy Rating: 10Wh)
```

Conclusion

In looking for solutions toward real-life problems, the implementation of Dynamic Programming, techniques and approaches, as well as the use of various data structures greatly help in the efficiency and operation in the intended purpose of the code.