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

```
# recursion method of sum of the squares of numbers in a list
def sqSumList(array, sum):
  if len(array) == 0:
    return sum
  else:
    num = array.pop()
    return sqSumList(array, sum + (num**2))

numbers = [2, 3, 4, 5]
sqSumList(numbers, 0)
```

```
54
```

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

```
# Dynamic Programing using Tables of the Sum of the squares of number in a list
def sqSumTable(array):
  table = [0] # Stores the Solution to lessen the time needed to solve the other iteration
  for i in array:
    num = array.pop()
    squared = num*num
    table[0] = squared + sqSumTable(array)
  return table[0]

numbers2 = [2, 3, 4, 5]
sqSumTable(numbers2)
```

```
54
```

Question:

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

- ***Recursion solves a problem with the blockers or conditional ending statements and calling its own function to solve a problem while dynamic programming focus in the small problem first while storing the result somewhere. This is crucial since recursion solves the problem repeatedly until the desired output but Dynamic programming solves the problem by storing the currently solved problem to a cache or a table that makes the process faster***

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

```
# bottom up is memoization: dict
# Memoization of a Triangular Sequence
tempDict = {}
def geoDict(num, memo):
  if num == 0:
    return 0
  if num not in memo:
    memo[num] = num + geoDict(num-1, memo)
  return memo[num]

geoDict(10, tempDict)
for i in tempDict:
  print(tempDict[i])
```

```
1
3
6
10
15
21
28
36
45
55
```

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

```
# Tops down uses tabulation:tables
# Tabulation of a Triangular Numbers
def SumTable(num):
  table = [0, 1]
  for i in range(2, num+1):
    table.append(i + table[i-1])
  return table

SumTable(10)
```

```
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
```

Question:

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

Type your answer here: ***Bottom up uses cache in the form of Dictionaries while Top Down uses tabulation methods in the form of Arrays***

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

Type your answer here.

Seatwork 2.1

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

```
#type your code here
#Recursion
def recursive_knap(budget ,capacity, weight, price, value, n):
  if n == 0 or capacity == 0:
    return 0

  if(weight[n-1] > capacity) or (price[n-1] > budget):
    return recursive_knap(budget, capacity, weight, price, value, n-1)

  else:
    return max(
        val[n-1] + recursive_knap(budget-price[n-1], capacity-weight[n-1], weight, price , value, n-1),
        recursive_knap(budget, capacity, weight, price, value, n-1)
    )

#Dynamic - Table
def tabulation_knap(budget, capacity, weight, price, val, n):
  #create the table
  table = [[0 for x in range(capacity+1)] for x in range (n+1)]
  table2 = [[0 for x in range(budget+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(capacity+1):
      if i == 0 or w == 0:
        table[i][w] = 0
      elif weight[i-1] ≤ w:
          table[i][w] = max(val[i-1] + table[i-1][w-weight[i-1]], table[i-1][w])

  for i in range(n+1):
    for w in range(budget+1):
      if i == 0 or w == 0:
        table2[i][w] = 0
      elif weight[i-1] ≤ w:
          table2[i][w] = max(val[i-1] + table2[i-1][w-price[i-1]], table2[i-1][w])
  return table[n][capacity], table2[n][budget]
```

RESULTS FOR ADDED BUDGET CRITERION

```
#To test:
val = [60, 100, 140, 180, 220] #values for the items
wt = [10, 20, 30, 40, 50] #weight of the items
prc = [50, 40, 30, 20, 10] # price of the items
cap = 100 #knapsack weight capacity
budget = 100 #knapsack budget/ price capacity
n = len(val) #number of items
```

```
recursive_knap(budget, cap, wt, prc, val, n) # Recursive
```

```
460
```

```
tabulation_knap(budget, cap, wt, prc, val, n) # Tabulation
```

```
(480, 640)
```

```
val1 = [60, 100, 140, 180, 220] #values for the items
wt1 = [10, 20, 30, 40, 50] #weight of the items
prc1 = [50, 40, 30, 20, 10] # price of the items
cap1 = 100 #knapsack weight capacity
budget1 = 100 #knapsack budget/ price capacity
n1 = len(val) #number of items

#initialize the container for the values that have to be stored
#values are initialized to -1
calc1 =[[-1 for i in range(cap+1)] for j in range(n+1)]
calc2 =[[-1 for i in range(budget+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 calc1[n][w] != -1:
    return calc1[n][w]

  #compute for the other cases
  if wt[n-1] <= w:
    calc1[n][w] = max(val[n-1] + mem_knapSack(wt, val, w-wt[n-1], n-1),
                      mem_knapSack(wt, val, w, n-1))
    return calc1[n][w]
  elif wt[n-1] > w:
    calc1[n][w] = mem_knapSack(wt, val, w, n-1)
    return calc1[n][w]

def mem_knapSackBud(wt, val, w, n):
  #base conditions
  if n == 0 or w == 0:
    return 0
  if calc2[n][w] != -1:
    return calc2[n][w]

  #compute for the other cases
  if wt[n-1] <= w:
    calc2[n][w] = max(val[n-1] + mem_knapSackBud(wt, val, w-wt[n-1], n-1), mem_knapSackBud(wt, val, w, n-1))
    return calc2[n][w]
  elif wt[n-1] > w:
    calc2[n][w] = mem_knapSackBud(wt, val, w, n-1)
    return calc2[n][w]
```

```
mem_knapSack(wt1, val1, cap1, n1)
```

```
480
```

```
mem_knapSackBud(prc1, val1, budget, n1)
```

```
640
```

Fibonacci Numbers

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

```
#type your code here
def fibTab(n):
  tb = [0, 1]
  for i in range(2, n+1):
    tb.append(tb[i-1] + tb[i-2])
  return tb

print(fibTab(6))
```

```
[0, 1, 1, 2, 3, 5, 8]
```

Supplementary Problem (HOA 2.1 Submission):

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

PROBLEM

- Mr. Jujutsu Martinez wants to limit his electrical bill to P2000/month while still maximizing the hours of usage of his appliances per day. With the given data, help him before he gives you a 3.00

all the data is referenced in this article: http://www.bulacanliving.com/electricity-cost-of-appliances.html

```
appliances = ["7 cu.ft refrigerator (non-inverter type)", "10kg automatic washing machine", "regular 16 inch desk fan", "regular 1.0 H.P airconditioner", "vacuum cleaner"]
time_usage = [24, 1, 20, 8, 1] # hrs/per day
pricess = [500, 530, 480, 1,900, 250] # rounded price per month
budgets = 2000 # in pesos
no = len(appliances) # No of appliances
```

Recursive

```
#type your code here for recursion programming solution
def recursion_solution(budget, price, time_usage, n):
  if n == 0 or budget == 0:
    return 0
  if(price[n-1] > budget):
    return recursion_solution(budget, price, time_usage, n-1)
  else:
    return max(time_usage[n-1] + recursion_solution(budget-price[n-1], price, time_usage, n-1),recursion_solution(budget, price, time_usage, n-1))
```

TABULATION METHOD

```
#type your code here for dynamic programming solution
def tabulation_solution(budget, price, time_usage, n):
  table = [[0 for x in range(budget+1)] for x in range (n+1)]

  for i in range(n+1):
    for w in range(budget+1):
      if i == 0 or w == 0:
        table[i][w] = 0
      elif price[i-1] ≤ w:
        table[i][w] = max(time_usage[i-1] + table[i-1][w-price[i-1]],table[i-1][w])
  return table[n][budget]
```

MEMOIZATION SOLUTION

```
memo =[[-1 for i in range(budgets+1)] for j in range(no+1)]

def memoi_solution(prices, time_usage, bt, n):
  if n == 0 or bt == 0:
    return 0
  if memo[n][bt] != -1:
    return memo[n][bt]
  if prices[n-1] <= bt:
    memo[n][bt] = max(time_usage[n-1] + memoi_solution(prices, time_usage, bt-prices[n-1], n-1), memoi_solution(prices, time_usage, bt, n-1))
    return memo[n][bt]
  elif prices[n-1] > bt:
    memo[n][bt] = memoi_solution(prices, time_usage, bt, n-1)
    return memo[n][bt]
```

Conclusion

```
#type your answer here
print("Recursion: ", recursion_solution(budgets, pricess, time_usage, no)) # ANSWER in RECURSIVE APPROACH
print("Tabulation: ", tabulation_solution(budgets, pricess, time_usage, no)) # ANSWER in TABULATION APPROACH
print("Memoization: ", memoi_solution(pricess, time_usage, budgets, no)) # ANSWER in MEMOIZATION APPROACH
print(f"It means that Mr. Jujutsu can use {memoi_solution(pricess, time_usage, budgets, no)} hrs of eletricity (Excluded the Aircon) without exceeding P2000 in electrical bill")
```

```
Recursion:  53
Tabulation:  53
Memoization:  53
It means that Mr. Jujutsu can use 53 hrs of eletricity (Excluded the Aircon) without exceeding P2000 in electrical bill
```

```
# Created by Kynamittens
```