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

```
# solves for the sum of all natural numbers up to n using recursion

def add_to_n(n):
  if n < 0:
    return 0
  else:
    return n + add_to_n(n-1)

test = add_to_n(5)
print(test)
```

```
15
```

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

```
# solves for the sum of all natural numbers up to n using dynamic programming

def add_to_n(n, memo):
  if n < 0:
    return 0
  if n not in memo:
    memo[n] = n + add_to_n(n-1, memo)
  return memo[n]

memo = {}
test = add_to_n(5, memo)
print(test)
```

```
15
```

Question:

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

Recursion differs from dynamic programming in the sense that dynamic programming, specifically memoization, requires the use of recursion in order for it to be performed. In dynamic programming, we save the data that we compute every time we recurse, removing the need to re-compute a value again if it needs to be repeated. In contrast, Recursion does the same operation but without saving the previously computed data.

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

```
# solves for the sum of all natural numbers up to n using tabulation

def add_to_n(n):
  nat_nums = [0,1]
  for i in range(2, n+1):
    nat_nums.append(i + nat_nums[i-1])
  return nat_nums

test = add_to_n(5)
for i in test:
  print(i)
```

```
0
1
3
6
10
15
```

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

```
# solves for the sum of all natural numbers up to n using memoization

def add_to_n(n, memo):
  if n < 0:
    return 0
  if n not in memo:
    memo[n] = n + add_to_n(n-1, memo)
  return memo[n]

memo = {}
test = add_to_n(5, memo)
for i in memo:
  print(memo[i])
```

```
0
1
3
6
10
15
```

Question:

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

The difference between bottom-up aand top-down dynamic programming comes down to the method that they use and how the data is being built up. Bottom-up dynamic programming utilizes tabulation, which uses an array to store values and only requires iteration. Meanwhile, top-down dynamic programming uses memoization, which utilizes a dictionary to store the computed data. Bottom-up, as the name suggests, starts off at nothing, building up to the desired solution. Top-down instead starts at the intended solution, working its way down to the other values needed.

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)
```

```
9950
```

```
#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)
```

```
10375
```

Code Analysis

With recursion, any previous data checked is not being stored, rather the optimal solution is just being computed. Any computation that isnt the solution will be disregarded, hence why there are 2 initial conditions that essentially skip being compared to a previous value.

With bottom-up dynamic programming, a 2-dimensional array is being used to store the data. Using similar conditions used in recursion, the array is being filled up until it is complete, and the solution is placed at the end of the array where it can be easily taken.

With top-down dynamic programming, a similar approach to bottom-up is done where a 2-dimensional array is used. However, the array is already filled up with -1 values, which are then being filled up by recursion, which was a method earlier. It starts at the end of the array and works its way down to the start of it.

Seatwork 2.1

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

```
#type your code here
#Recursion



#Dynamic



#Memoization
```

Supplementary Problem (HOA 2.1 Submission):

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

```
"""A logistics company is acting as a sub-contractor for a famous online shopping site.
Lei is a part-time employee at the logistics company, employed as a delivery rider for
parcels at a warehouse. Today, there are 10 different parcels to be delivered, each with
their own individual locations, weights, and values. Before he needs to go to school,
Lei wants to determine the parcels that have the maximum price value in a given set of time."""

#Recursion Solution
def recKnapSack(maxTime, times, values, n):
  if n == 0 or maxTime == 0:
    return 0
  if(times[n-1] > maxTime):
    return recKnapSack(maxTime, times, values, n-1)
  else:
    return max(values[n-1] + recKnapSack(maxTime-times[n-1], times, values, n-1),
            recKnapSack(maxTime, times, values, n-1))

prices = [100, 1200, 200, 900, 2300, 650, 1950, 525, 225, 3500]
times = [45, 30, 60, 60, 75, 120, 90, 30, 105, 20]
weights = [10, 500, 400, 20, 2500, 200, 2000, 100, 30, 4000]
maxTime = 335
n = len(prices)

print(f"Maximum Price Value possible within 240 minutes: {recKnapSack(maxTime, times, prices, n)}")
print(f"Maximum Weight Value possible within 240 minutes: {recKnapSack(maxTime, times, weights, n)}")
```

```
Maximum Price Value possible within 240 minutes: 10375
Maximum Weight Value possible within 240 minutes: 9500
```

```
#Memoization Solution
prices = [100, 1200, 200, 900, 2300, 650, 1950, 525, 225, 3500]
times = [45, 30, 60, 60, 75, 120, 90, 30, 105, 20]
weights = [10, 500, 400, 20, 2500, 200, 2000, 100, 30, 4000]
maxTime = 335
n = len(prices)

memo_prices = [[-1 for i in range(maxTime+1)] for j in range(n+1)]
memo_weights = [[-1 for i in range(maxTime+1)] for j in range(n+1)]
calc = [[-1 for i in range(maxTime+1)] for j in range(n+1)]

def memKnapSack(maxTime, times, values, memo, n):
  if n == 0 or maxTime == 0:
    return 0
  if memo[n][maxTime] != -1:
    return memo[n][maxTime]

  #compute for the other cases
  if times[n-1] <= maxTime:
    memo[n][maxTime] = max(values[n-1] + memKnapSack(maxTime-times[n-1], times, values, memo, n-1),
                      memKnapSack(maxTime, times, values, memo, n-1))
    return memo[n][maxTime]
  elif times[n-1] > maxTime:
    memo[n][maxTime] = memKnapSack(maxTime, times, values, memo, n-1)
    return memo[n][maxTime]

print(f"Maximum Price Value possible within 240 minutes: {memKnapSack(maxTime, times, prices, memo_prices, n)}")
print(f"Maximum Weight Value possible within 240 minutes: {memKnapSack(maxTime, times, weights, memo_weights, n)}")
```

```
Maximum Price Value possible within 240 minutes: 10375
Maximum Weight Value possible within 240 minutes: 9500
```

```
#Tabular Solution
def tabKnapSack(maxTime, times, values, n):
  table = [[0 for x in range(maxTime+1)] for x in range (n+1)]

  for n in range(n+1):
    for maxTime in range(maxTime+1):
      if n == 0 or maxTime == 0:
        table[n][maxTime] = 0
      elif times[n-1] <= maxTime:
        table[n][maxTime] = max(values[n-1] + table[n-1][maxTime-times[n-1]],
                        table[n-1][maxTime])
  return table[n][maxTime]


prices = [100, 1200, 200, 900, 2300, 650, 1950, 525, 225, 3500]
```

```
prices = [100, 1200, 200, 900, 2300, 650, 1950, 325, 225, 3500]
times = [45, 30, 60, 60, 75, 120, 90, 30, 105, 20]
weights = [10, 500, 400, 20, 2500, 200, 2000, 100, 30, 4000]
maxTime = 335
n = len(prices)

print(f"Maximum Price Value possible within 240 minutes: {tabKnapSack(maxTime, times, prices, n)}")
print(f"Maximum Weight Value possible within 240 minutes: {tabKnapSack(maxTime, times, weights, n)}")

```

```
Maximum Price Value possible within 240 minutes: 10375
Maximum Weight Value possible within 240 minutes: 9500
```

Conclusion

While the Knapsack Problem can be solved using the Bruteforce and Greedy Methods, it can also be solved using Recursion and Dynamic