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

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

Question:

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

Type your answer here: it came from it words the bottom-up is where you write your code of the values where it is in the top of your code while the top-down is the reverse of the bottom-up that uses the code to enumerate the values from the bottom

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)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-a876c5ea3b2a> in <cell line: 7>()
      5 n = len(val) #number of items
      6
----> 7 rec_knapSack(w, wt, val, n)

                                 2 frames
<ipython-input-12-78cb17c4648a> in rec_knapSack(w, wt, val, n)
     18   # (2) don't include the nth item
     19   else:
---> 20     return max(
     21         val[n-1] + rec_knapSack(
     22             w-wt[n-1], wt, val, n-1),

TypeError: 'int' object is not callable
```

SEARCH STACK OVERFLOW

```
#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

First to do is create a table and create if its bottom-up or top-down base on the conditions and the nth value is the length of the values

Seatwork 2.1

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

```
#type your code here
#Recursion
#sample code for knapsack problem using recursion
def jolibee(maxs, weight, price, n):


  if n == 0 or maxs == 0:
    return 0

  if(weight[n-1] > maxs):
    return jolibee(w, wt, val, n-1)

#diko na po ma solve out of time sorry
#nag kaka problem sa else statement
  else:
    return max(
        price[n-1] + jolibee(
            maxs-weight[n-1], weight, price, n-1),
            jolibee(maxs, weight, price, n-1)
    )

max1 = int(input(f"Enter the Maximum weight of the item: "))
items = int(input("Enter the number of items: "))
weight1 = []
price1 = []
for i in range(items):
  weight = int(input(f"Enter the weight of the item {i+1}:"))
  weight1.append(weight)
  price = int(input(f"Enter the price of the item {i+1}:"))
  price1.append(price)


n1 = len(price1)
jolibee(max1, weight1, price1, n1)
```

```
#Dynamic
```

```
#Memoization
```

```
Enter the Maximum weight of the item: 50
Enter the number of items: 2
Enter the weight of the item 1:30
Enter the price of the item 1:400
Enter the weight of the item 2:10
Enter the price of the item 2:20
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-c649d4f1bfeb> in <cell line: 32>()
     30
     31 n1 = len(price1)
---> 32 jolibee(max1, weight1, price1, n1)
     33
     34

                              1 frames
<ipython-input-19-c649d4f1bfeb> in jolibee(maxs, weight, price, n)
     12
     13   else:
---> 14     return max(
     15         price[n-1] + jolibee(
     16             maxs-weight[n-1], weight, price, n-1),

TypeError: 'int' object is not callable
```

SEARCH STACK OVERFLOW

```
#Dynamic
```

```
#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]


w = input(f"Enter the Maximum weight of the item: ")
items = int(input("Enter the number of items: "))
wt = []
val = []
for i in range(items):
  weight = input(f"Enter the weight of the item {i+1}:")
  wt.append(weight)
  price = input(f"Enter the price of the item {i+1}:")
  val.append(price)


n = len(val)

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

```
Enter the Maximum weight of the item: 50
Enter the number of items: 3
Enter the weight of the item 1:10
Enter the price of the item 1:60
Enter the weight of the item 2:20
Enter the price of the item 2:100
Enter the weight of the item 3:30
Enter the price of the item 3:120
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-deebaacf8894> in <cell line: 32>()
     30 n = len(val)
     31
---> 32 DP_knapSack(w, wt, val, n)

    1 frames
<ipython-input-24-deebaacf8894> in <listcomp>(.0)
      4 def DP_knapSack(w, wt, val, n):
      5   #create the table
----> 6   table = [[0 for x in range(w+1)] for x in range (n+1)]
      7
      8   #populate the table in a bottom-up approach

TypeError: can only concatenate str (not "int") to str
```

SEARCH STACK OVERFLOW

```
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)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-25-2d63d04fddb6> in <cell line: 27>()
     25     return calc[n][w]
     26
---> 27 mem_knapSack(wt, val, w, n)

    2 frames
<ipython-input-25-2d63d04fddb6> in mem_knapSack(wt, val, w, n)
     18   #compute for the other cases
     19   if wt[n-1] <= w:
---> 20     calc[n][w] = max(val[n-1] + mem_knapSack(wt, val, w-wt[n-1], n-1),
     21                      mem_knapSack(wt, val, w, n-1))
     22     return calc[n][w]

TypeError: 'int' object is not callable
```

SEARCH STACK OVERFLOW

Fibonacci Numbers

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

```
#type your code here
```

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
```

```
#type your code here for dynamic programming solution
```

Conclusion

```
#type your answer here
```