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

```
def sum_of_digits(n):
  if n < 10:
    return n
  else:
    return n % 10 + sum_of_digits(n//10)
number = int(input("Enter mulitple digits to get the sum: "))
sum_of_digits(number)
```

```
Enter mulitple digits to get the sum: 12345
15
```

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

```
def sum_of_digits(n, memo={}):
  if n in memo:
    return memo[n]
  if n < 10:
    return n
  else:
    result = n % 10 + sum_of_digits(n//10, memo)
    memo[n] = result
    return result
number = int(input("Enter mulitple digits to get the sum: "))
sum_of_digits(number)
```

```
Enter mulitple digits to get the sum: 12345
```

```
15
```

Question:

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

While both techniques employ recursion to solve the same issue, dynamic programming with memoization improves the solution by eliminating unnecessary computations, resulting in shorter execution times and more efficiency.

Double-click (or enter) to edit

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

```
def sum_of_digits_bottom_up(n):
    dp = [0] * (n + 1)
    for i in range(10):
        dp[i] = i
    for i in range(10, n + 1):
        dp[i] = i % 10 + dp[i // 10]

    return dp[n]
number = int(input("Enter mulitple digits to get the sum: "))
sum_of_digits_bottom_up(number)
```

```
Enter mulitple digits to get the sum: 12345
15
```

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

```
def sum_of_digits_top_down(n, memo={}):
    if n in memo:
        return memo[n]
    if n < 10:
        return n
    else:
        result = n % 10 + sum_of_digits_top_down(n // 10, memo)
        memo[n] = result
        return result
number = int(input("Enter mulitple digits to get the sum: "))
sum_of_digits_top_down(number)
```

```
Enter mulitple digits to get the sum: 12345
15
```

Question:

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

While both approaches seek to solve problems efficiently by breaking them down into smaller subproblems, bottom-up begins with the base case and iteratively progresses to the desired solution, whereas top-down begins with the larger problem and recursively breaks it down into smaller subproblems, storing the results of subproblems to avoid redundant calculations.

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
```

```
#Dynamic
```

```
#Memoization
```

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

Problem: Triangular numbers are generated by adding successive natural integers. For instance, the first few triangular numbers are 1, 3 (1+2), 6 (1+2+3), 10 (1+2+3+4), and so forth.

```
#type your code here for recursion programming solution
def triNum_recursive(n):
    if n == 1:
        return 1
    else:
        return n + triNum_recursive(n - 1)
num = int(input("Enter a number to find the triangular numbers: "))
print(triNum_recursive(num))
```

```
Enter a number to find the triangular numbers: 5
15
```

```
#type your code here for dynamic programming solution
def triNum_dynamic(n):
    if n == 1:
        return 1
    memo = [-1] * (n + 1)
    memo[1] = 1
    return triNum_dynamic_helper(n, memo)

def triNum_dynamic_helper(n, memo):
    if memo[n] != -1:
        return memo[n]
    memo[n] = n + triNum_dynamic_helper(n - 1, memo)
    return memo[n]
num = int(input("Enter a number to find the triangular numbers: "))
print(triNum_dynamic(num))
```

```
Enter a number to find the triangular numbers: 5
15
```

Conclusion

In conclusion, my investigation of dynamic programming and recursion has provided me with a fundamental grasp of their concepts and applications in the solution of numerous computational issues. Dynamic programming has given me insight into improving solutions by making optimal use of memoization, resulting in quicker and more scalable algorithms. Furthermore, studying recursion has given me a better understanding of problem-solving methodologies, allowing me to divide large tasks into smaller subproblems and tackle them repeatedly. Overall, this voyage has extended my view on algorithmic design and problem-solving techniques, as well as improved my technical abilities.