Dejoras

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

```
n = int(input("Enter number of elements : "))

def list_sum(n):

    if n == 0: # if statement for list who have only one value

        return 0 # return that one value as it is the sum of the whole list
    else:
        ele = int(input("Enter a number: "))
        return ele + list_sum(n-1)

print(list_sum(n))
```

```
 Enter number of elements : 3
 Enter a number: 5
 Enter a number: 5
 Enter a number: 5
 15
```

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

```
# using bottom-up approach

n = int(input("Enter number of elements : "))

def sum_list(n):

    storage = []
    cumulative_sum = 0
    # iterating till the range
    for i in range(1, n+1):
      ele = int(input("Enter element: "))
      # adding the element
      cumulative_sum += ele
      storage.append(cumulative_sum)
    return storage

sum_list(n)
```

```
 Enter number of elements : 3
 Enter element: 5
 Enter element: 5
 Enter element: 5
 [5, 10, 15]
```

Question:

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

Using recursion is almost the same from using dynamic programming, as you do recursion in dynamic programming to store and reuse the solutions at the expense of expanding space complexity yet optimizes time complexity. The given sample codes focused on getting the sum of the elements inputted by the user. In recursion, the function is ran repeatedly until it reaches the input limit from the user while adding the total. In dynamic programming, smaller subproblems are solved up until to the bigger subproblems using the smaller subproblems that are solved before. That is why it has a lesser time complexity by recycling those subproblems. But for comparison of my two codes, instead of recursion, I used Iteration as I used the other approach of dynamic programming which is bottom-up approach.

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

```
# using bottom-up approach

n = int(input("Enter number of elements: "))

def prod_list(n):
    storage = []
    cumulative_product = 1

    for i in range(1, n + 1):
        ele = int(input("Enter element: "))
        cumulative_product *= ele
        storage.append(cumulative_product)

    return storage

prod_list(n)
```

```
 Enter number of elements: 3
 Enter element: 5
 Enter element: 5
 Enter element: 5
 [5, 25, 125]
```

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

```
def cube(n, ele, memo = {}):

    if n == 0:
      return 1

    if n == 1:
      return ele

    memo[n] = ele * cube(n - 1, ele, memo)
    print(f"Element at index {n}: {ele}")
    print(memo)
    return memo[n]

ele = int(input("Enter element: "))

Dict = {}

cube(3, ele, Dict)
```

```
 Enter element: 5
 Element at index 2: 5
 {2: 25}
 Element at index 3: 5
 {2: 25, 3: 125}
 125
```

Question:

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

Bottom-up programming and top-down programming are exact opposites. For this activity, I made the function of the two programs different where bottom-up would get the cumulative product while the top-up would calculate the cube of the given element. As mentioned earlier in explaining dynamic programming, smaller subproblems are solved up until to the bigger subproblems using the smaller subproblems that are solved before. This is the case for bottom-up approach as dynamic programming consists of two approaches. Top-down approach is the opposite of it as it recursively breaks down a problem into smaller subproblems. In this case, the solutions to the subproblems are stored in a dictionary to avoid redundant calculations. Thus, checking if a solution for a specific subproblem is already stored before recalculating it.

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

1. Recursion
   If the nth item or the capacity is equal to 0. It simply returns 0 as basically it has no value. The next if-statement is that if the weight of a certain object exceeds the capacity, that certain object would remain by returning its default value. Lastly, the else statement is for returning the maximum possible value using the max() function after calculating the possible objects that could be optimized uptil the maximum capacity. That is why the objects that have 20 and 30 as weight are used because 50 is the maximum capacity thus adding up their weights upto 220.
2. Dynamic Programming
   The table variable was initialized to 0 and is utilized in the table creation process. Next, a nested for loop is constructed, adding the weight capacity and number of objects for each iteration. If the item weight is less than the capacity, the nested for loop computes the total amount and returns the table; otherwise, the if condition inside the loop specifies that either i or w equates to 0.
3. Memoization
   Four parameters are used by the function of memoization. A memoization table is set with a value of -1 which is named as "calc". Same with the other two, if the capacity and object(n) is 0. It will return 0. The not equal statement (!=) implies that if n and w has already been stored, it returns that stored result to avoid redundant calculation. For the next if statement, it is the same process of getting the maximum value by using the max() function after calculating the possible objects that could be optimized uptil the maximum capacity. Consequently, the elif statement returns the value as is because the weight exceeds the maximum capacity.

Seatwork 2.1

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

```
#type your code here
#Recursion
def rec_knapSack(w, wt, val, c, 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, c, 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, c, n-1),
            rec_knapSack(w, wt, val, c, n-1)
    )

c = [250, 500, 100] #cost for the items
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, c, n)


#Dynamic
def DP_knapSack(w, wt, val, c, 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]

c = [250, 500, 100] #cost for the items
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

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



#Memoization
#initialize the list of items
c = [250, 500, 100]
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, c, 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], c, n-1),
                     mem_knapSack(wt, val, w, c, 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, c, n)
```

```
     220
```

Fibonacci Numbers

```
scale = int(input("Enter scale of elements in the fibonacci sequence: "))

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

    return storage

fibTabulation(scale)
```

```
 Enter scale of elements in the fibonacci sequence: 5
 [0, 1, 1, 2, 3, 5]
```

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

```
scale = int(input("Enter scale of elements in the fibonacci sequence: "))

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

    return storage


result = fibTabulation(scale)
def n_fibTabulation(result, n):
    nth_fibo = []
    print(result)
    for i in range(scale):
       nth_fibo.append(i)
    return nth_fibo

print("nth numbers of Fibonacci Series using Dynamic Programming")
n_fibTabulation(result, n)
```

```
 Enter scale of elements in the fibonacci sequence: 5
 [0, 1, 1, 2, 3]
 [0, 1, 2, 3, 4]
```

Supplementary Problem (HOA 2.1 Submission):

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

Recursion

This is a problem for fools who have trouble listing up all the desired objects they could carry yet displays recursion effectively. I call this tallying.

```
n = int(input("Enter number of limited capacity: "))

def tally(n):

    if n == 0:

        return ""
    else:
        objects = ""
        for i in range(n):
            m = str(input("Enter an object you want to carry: "))
            objects += m + " "

            return objects + str(tally(n-1))
result = tally(n)

print("The objects you would like to carry are: " , result)
```

```
 Enter number of limited capacity: 3
 Enter an object you want to carry: Hammer
 Enter an object you want to carry: Screwdriver
 Enter an object you want to carry: Facemask
 The objects you would like to carry are:  Hammer Screwdriver Facemask
```

Dynamic programming

```
n = int(input("Enter number of limited capacity: "))

def tally_bottom_up(n):
    objects_table = [""] * n

    for i in range(n):
        objects = ""
        m = str(input(f"Enter an object you want to carry for iteration {i + 1}: "))
        objects += m + " "
        objects_table[i] = objects

    return objects_table

resultz = tally_bottom_up(n)
resultz.sort()
print("The objects you would like to carry are:", resultz)
```

```
 Enter number of limited capacity: 3
 Enter an object you want to carry for iteration 1: Toothpaste
 Enter an object you want to carry for iteration 2: Toothbrush
```

```
Enter an object you want to carry for iteration 3: Dental floss
The objects you would like to carry are: ['Dental floss ', 'Toothbrush ', 'Toothpaste ']
```

Conclusion

After conducting this Activity, I gained insight on the fundamentals of Dynamic Programming specifically for its two approaches which are bottom-up and top-down approach. These two approaches operate on different ways yet serve the same purpose by avoiding redundant calculation of subproblems by storing them to a certain list or dictionary. This would worsen the space complexity of the program yet eases up the time complexity of it. In general, Dynamic Programming is beneficial when the problem exhibits optimal substructures, as it breaks down the problem into smaller overlapping subproblems, reducing redundancy and improving overall efficiency of the program.