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

In [1]:
```
def biCoefficientTab(n, k):
  tb = [[0] * (k+1) for _ in range(n+1)]
  for i in range(n+1):
    for j in range(min(i, k)+1):
      if j == 0 or j == i:
        tb[i][j] = 1
      else:
        tb[i][j] = tb[i-1][j-1] + tb[i-1][j]
  return tb[n][k]

n = 4
k = 2
print("Bottom-up Approach")
print("Value of C(%d,%d) is (%d)" % (n, k, biCoefficientTab(n, k)))
```

```
Bottom-up Approach
Value of C(4,2) is (6)
```

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

In [2]:
```
def biCoefficientMem(n, k, memo):
  if (n, k) in memo:
    return memo[n,k]
  if k > n:
    return 0
  if k == 0 or k == n:
    return 1
  memo[n, k] = biCoefficientMem(n-1, k-1, memo) + biCoefficientMem(n-1, k, mem
  return memo[n, k]

memo = {}
n = 4
k = 2
print("Top-down Approach")
print("Value of C(%d,%d) is (%d)" % (n, k, biCoefficientMem(n, k, memo)))
```

```
Top-down Approach
Value of C(4,2) is (6)
```

Question:

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

Type your answer here:

ANSWER:

- Bottom-up approach goes for the specifics first, that is why in memoization, we use dictionaries in order to store values that have already been solve or used before in order to just return rather than solving it again which saves time. Top-down approach on the other hand focuses on the general and makes it to the specific which we do by listing all possible results in the table and having to compute the solution with the use of the results we acquired in the table.

0/1 Knapsack Problem

- Analyze three different techniques to solve knapsacks problem

1. Recursion
2. Dynamic Programming
3. Memoization

In [ ]:
```
#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)
    )
```

In [ ]:
```
#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)
```

Out[27]: 220

In [ ]:
```
#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]
```

In [ ]:
```
#To test:
val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
n = len(val)

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

Out[29]:
```
220
```

In [ ]:
```
#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)
```

Out[31]: 220

Code Analysis

Type your answer here.

Seatwork 2.1

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

Fibonacci Numbers

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

In [ ]:
```
# memoization
def fibMemo(n, memo):
  if n == 1:
    return 0
  if n == 2:
    return 1
  if not n in memo:
      memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo)
  return memo[n]

tempDict = {}
fibMemo(6, tempDict)

print("Fibonacci Series - Memoization")
print("0")
print("1")
for element in tempDict.values():
    print(element)
```

```
Fibonacci Series - Memoization
0
1
1
2
3
5
```

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

print("Fibonacci Series - Tabulation")
print(fibTab(6))
```

```
Fibonacci Series - Tabulation
[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

- You are in a bookstore having a specific amount of money. The goal is to pick however number of books according to the given constraint (budget) that contains the best ratings/review (1-5).

In [28]:
```
# recursive solution
def knapRecursion(capacity, prices, ratings, n):
    if n == 0 or capacity == 0:
        return []
    if prices[n - 1] > capacity:
        return knapRecursion(capacity, prices, ratings, n - 1)
    else:
        bought_books = knapRecursion(capacity - prices[n - 1], prices, ratings
        not_bought_books = knapRecursion(capacity, prices, ratings, n - 1)
        if sum(book[1] for book in bought_books) > sum(book[1] for book in not
            return bought_books
        else:
            return not_bought_books
```

In [37]:
```
def totalRecursive(total_price, total_rating):
  for book_index, rating in books:
      total_price += prices[book_index - 1]
      total_rating += rating
      print(title[book_index - 1], "- Rating:", rating, "- Price:", prices[boo
  print("\nTotal Price:", total_price)
  print("Total Rating:", total_rating)
```

In [38]:
```
title = ['book1', 'book2', 'book3', 'book4', 'book5','book6', 'book7', 'book8'
ratings = [2, 5, 3, 4.5, 5, 4, 2, 5]
prices = [599, 650, 600, 435, 550, 499, 399, 499]
budget = 2000

total_books = len(title)
books = knapRecursion(budget, prices, ratings, total_books)

total_price = 0
total_rating = 0

print("Books to buy:")
totalRecursive(total_price, total_rating)
```

```
Books to buy:
book4 - Rating: 4.5 - Price: 435
book5 - Rating: 5 - Price: 550
book6 - Rating: 4 - Price: 499
book8 - Rating: 5 - Price: 499

Total Price: 1983
Total Rating: 18.5
```

In [42]:
```
# dynamic solution (tabulation)
def knapDynamic(budget, prices, ratings):
    n = len(prices)
    dp = [[0] * (budget + 1) for _ in range(n + 1)]
    bought_books = [[[] for _ in range(budget + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(1, budget + 1):
            if prices[i - 1] <= j:
                included = ratings[i - 1] + dp[i - 1][j - prices[i - 1]]
                not_bought = dp[i - 1][j]
                if included > not_bought:
                    dp[i][j] = included
                    bought_books[i][j] = bought_books[i - 1][j - prices[i - 1]
                else:
                    dp[i][j] = not_bought
                    bought_books[i][j] = bought_books[i - 1][j]
            else:
                dp[i][j] = dp[i - 1][j]
                bought_books[i][j] = bought_books[i - 1][j]

    return bought_books[n][budget]
```

In [43]:
```
def totalDynamic(total_price, total_rating):
  for i, rating in books:
        total_price += prices[i - 1]
        total_rating += rating
        print(title[i - 1], "- Rating:", rating, "- Price:", prices[i - 1])
  print("\nTotal Price:", total_price)
  print("Total Rating:", total_rating)
```

In [44]:
```
title = ['book1', 'book2', 'book3', 'book4', 'book5','book6', 'book7', 'book8'
ratings = [2, 5, 3, 4.5, 5, 4, 2, 5]
prices = [599, 650, 600, 435, 550, 499, 399, 499]
budget = 2000  # Change the budget here as needed

books = knapDynamic(budget, prices, ratings)
total_price = 0
total_rating = 0

print("Books to buy:")

totalDynamic(total_price, total_rating)
```

```
Books to buy:
book4 - Rating: 4.5 - Price: 435
book5 - Rating: 5 - Price: 550
book6 - Rating: 4 - Price: 499
book8 - Rating: 5 - Price: 499

Total Price: 1983
Total Rating: 18.5
```

Conclusion

In [ ]:
```
#type your answer here
```

- This activity introduced dynamic programming with the use of memoization and tabulation, a top-down and bottom-up approach in programing. For the first task, I implemented binomial coefficients in both techniques which I found to understand after looking up some examples and actually applying it to my code. I'm specially fond of the memoization method for it stores data that has already been computed to avoid having to work it all out again. The tabulation method is also handy but I still get confuse on how the tables are being implemented and how to properly create them. For the knapsack problem, I used the same real world problem that I created for the last activity. Converting it to possess a recursive quality is quite tricky and I'm still kind of confused on how it work but I think I got the output right. In the tabulation version of it, I just followed the original code I made using brute force algorithm but the creation and utilization took quite some time to make and work properly. All in all I still need to practice more in order to fully grasp the subject and actually implement it in which it makes sense to me.