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:

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 [64]:
```
#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[64]: 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 [56]:

```python
#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[56]: 220

Code Analysis

Type your answer here.

Seatwork 2.1

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

In [85]:

```python
#Recursion
val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
n = len(val)
tax = 10


def recur(w, wt, val, n, ta):
  if n == 0 or w == 0:
    return 0
  if(wt[n-1] > w):
    return recur(w, wt, val, n-1)
  else:
    return max(
        val[n-1] + recur(
            w-wt[n-1], wt, val, n-1),
            recur(w, wt, val, n-1)
    )
#Dynamic
def dp(w, wt, val, n):

  table = [[0 for x in range(w+1)] for x in range (n+1)]


  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]

#Memoization
calc =[[-1 for i in range(w+1)] for j in range(n+1)]


def memoi(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] + memoi(wt, val, w-wt[n-1], n-1),
                     memoi(wt, val, w, n-1))
    return calc[n][w]
  elif wt[n-1] > w:
    calc[n][w] = memoi(wt, val, w, n-1)
    return calc[n][w]
```

In [86]:
```
print("recursion\n",recur(w, wt, val, n))
print("dynamic progamming\n",dp(w, wt, val, n))
print("memoization\n",memoi(wt, val, w, n))
```

```
recursion
 220
dynamic progamming
 220
memoization
 220
```

Fibonacci Numbers

In [ ]:
```

```

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

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

Supplementary Problem (HOA 2.1 Submission):

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

In [ ]:
```
#type your code here for recursion programming solution
```

In [ ]:
```
#type your code here for dynamic programming solution
```

Conclusion

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