Lab 7: Dynamic Programming

Lab associated with Module 7: Dynamic Programming

In [ ]:
```
# The following lines are used to increase the width of cells to utilize more sp
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:95% !important; }</style>"))
```

Section 0: Imports

In [ ]:
```
import numpy as np
```

In [ ]:
```
import math
```

In [ ]:
```
from IPython.display import Image
from graphviz import Digraph
```

Details of Digraph package: https://h1ros.github.io/posts/introduction-to-graphviz-in-jupyter-notebook/

Activity 1: You are running up a staircase with a total of n steps. You can hop either 1 step, 2 steps or 3 steps at at time. Write a DP program to determine how many possible ways you can run up the stairs? (Hint: Start with a recursive solution, and then later move to top-down approach of DP).

In [ ]:
```
def Recursive_Stairs(n):
    """Recursive version of Stairs. No Memoization"""
    """Args: n: number of stairs"""
    """Returns: number of ways to climb n stairs"""

    # Base cases
    if n < 0:
        return 0
    elif n == 0:
        return 1
    else:
        # Recursively call the function
        return Recursive_Stairs(n-1) + Recursive_Stairs(n-2) + Recursive_Stairs(

print(Recursive_Stairs(10))

def Memo_Stairs(n):
    """Memoization version of Recursive_Stairs"""
    """Args: n: number of stairs"""
    """Returns: number of ways to climb n stairs"""
```

```
    # Create a memo list
    memo = [0] * (n+1)
    return Memo_Stairs_Helper(n, memo)

def Memo_Stairs_Helper(n, memo):
    """Helper function for Memo_Stairs"""
    """Args: n: number of stairs"""
    """Retruns: memo: list of memoized values"""
    # Base cases
    if n < 0:
        return 0
    elif n == 0:
        return 1
    # If the value is already in the memo, return it
    elif n in memo:
        return memo[n]
    else:
        # Recursively call the helper function
        # Add the result to the memo
        memo[n] = Memo_Stairs_Helper(n-1, memo) + Memo_Stairs_Helper(n-2, memo)
        return memo[n]

print(Memo_Stairs(10))
```

Activity 2: Write the code for finding the Longest Common Sub-sequence. Make sure you output the Matrix C and the longest sub-sequence as well. Test your code with various use-cases.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + C[i\text{-}1, j\text{-}1] & \text{if } X[i] = Y[i] \\ \max C[i, j\text{-}1], C[i\text{-}1,j] & \text{if } X[i] \mathrel{!=} Y[i] \end{cases}$$

In [ ]:

```
import itertools
def Longest_SS(X, Y):
    """Longest subsequence of X and Y"""
    """Args: X and Y are strings"""
    """Returns: the longest subsequence of X and Y"""

    # Initialize the matrix to the length of the strings
    C = [[0 for _ in range(len(Y)+1)] for _ in range(len(X)+1)]
    # Initialize the first row and column to 0
    for i in range(len(X)+1):
        C[i][0] = 0
    for j in range(len(Y)+1):
        C[0][j] = 0

    # Recursive solution
    for i, j in itertools.product(range(1, len(X)+1), range(1, len(Y)+1)):
        # Either update the matrix or choose from a value that we have already c
        C[i][j] = C[i-1][j-1] + 1 if X[i-1] == Y[j-1] else max(C[i-1][j], C[i][j
```

```
    # Backtrack to find the subsequence
    # Start at the bottom right corner of the matrix
    i, j = len(X), len(Y)
    subseq = []
    while i > 0 and j > 0:
        # Move diagonally if the characters match
        if X[i-1] == Y[j-1]:
            subseq.append(X[i-1])
            i -= 1
            j -= 1
        # Otherwise, move in the direction of the larger value
        elif C[i-1][j] > C[i][j-1]:
            i -= 1
        else:
            j -= 1

    print(''.join(reversed(subseq)))

    for row in C:
        print(row)

    return ''.join(reversed(subseq))
```

In [ ]:

```
import unittest

class TestLongestSS(unittest.TestCase):
    def test_Longest_SS(self):
        # Test case 1: X and Y have no common subsequence
        X = "ABC"
        Y = "DEF"
        expected_output = ""
        print("\nTest case 1: X and Y have no common subsequence")
        self.assertEqual(Longest_SS(X, Y), expected_output)

        # Test case 2: X and Y have a common subsequence of length 1
        X = "ABC"
        Y = "BCD"
        expected_output = "BC"
        print("\nTest case 2: X and Y have a common subsequence of length 2")
        self.assertEqual(Longest_SS(X, Y), expected_output)

        # Test case 3: X and Y have a common subsequence of length 3
        X = "ABCBDAB"
        Y = "BDCAB"
        expected_output = "BDAB"
        print("\nTest case 3: X and Y have a common subsequence of length 4")
        self.assertEqual(Longest_SS(X, Y), expected_output)

        # Test case 4: X and Y are identical
        X = "ABC"
        Y = "ABC"
        expected_output = "ABC"
        print("\nTest case 4: X and Y are identical and have a comm length 3")
        self.assertEqual(Longest_SS(X, Y), expected_output)

        # Test case 5: X and Y have different lengths
        X = "ABC"
        Y = "ABCD"
        expected_output = "ABC"
        print("\nTest case 5: X and Y have different lengths and the subsequence
```

```
        self.assertEqual(Longest_SS(X, Y), expected_output)

# Create a test suite
suite = unittest.TestLoader().loadTestsFromTestCase(TestLongestSS)

# Run the test suite and print the results
runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

```
test_Longest_SS (__main__.TestLongestSS.test_Longest_SS) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Test case 1: X and Y have no common subsequence

[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]

Test case 2: X and Y have a common subsequence of length 2
BC
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 1, 1, 1]
[0, 1, 2, 2]

Test case 3: X and Y have a common subsequence of length 4
BDAB
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 1]
[0, 1, 1, 1, 1, 2]
[0, 1, 1, 2, 2, 2]
[0, 1, 1, 2, 2, 3]
[0, 1, 2, 2, 2, 3]
[0, 1, 2, 2, 3, 3]
[0, 1, 2, 2, 3, 4]

Test case 4: X and Y are identical and have a comm length 3
ABC
[0, 0, 0, 0]
[0, 1, 1, 1]
[0, 1, 2, 2]
[0, 1, 2, 3]

Test case 5: X and Y have different lengths and the subsequence has length 3
ABC
[0, 0, 0, 0, 0]
[0, 1, 1, 1, 1]
[0, 1, 2, 2, 2]
[0, 1, 2, 3, 3]
```

Out[ ]:
```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

Section 2: Unbounded Knapsack Problem

Let us build a solution to unbounded Knapsack problem.

In [ ]:
```
def unboundedKnapsack(W, n, wt, vals, names):

    K = [0 for _ in range(W + 1)]
    ITEMS = [[] for _ in range(W + 1)]

    for x in range(1, W + 1):
        K[x] = 0
        for i in range(1, n):

            prev_k = K[x]

            if (wt[i] <= x):
                K[x] = max(K[x], K[x - wt[i]] + vals[i])

            if K[x] != prev_k:
                ITEMS[x] = ITEMS[x - wt[i]] + names[i]


    return K[W], ITEMS[W]
```

In [ ]:
```
W = 4
wt = [1, 2, 3]
vals = [1, 4, 6]
names = [["Turtle"], ["Globe"], ["WaterMelon"]]

n = len(names)

print(f'We have {n} items')
```

In [ ]:
```
K, ITEMS = unboundedKnapsack(W, n, wt, vals, names)
```

In [ ]:
```
ITEMS
```

Activity 3: In the earlier activity, you analysed the code for unbounded knapsack. Based on the algorithm discussed in this section, implement a solution to do 0/1 Knapsack. Make sure you test your algorithms for various test-cases.

In [ ]:
```
import itertools
def Zero_One_Knapsack(n, wt, vals, W):
    # make a matrix of size W+1 x n+1
    K = [["." for _ in range(W+1)] for _ in range(n+1)]
    # initialize the first row and column to 0
    for i in range(n+1):
        K[i][0] = 0
    for j in range(W+1):
        K[0][j] = 0
    # fill in the rest of the matrix
    for i, w in itertools.product(range(1, n+1), range(1, W+1)):
        K[i][w] = (
            max(vals[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w])
            if wt[i - 1] <= w
```

```
            else K[i - 1][w]
        )
    # print the matrix
    for row in K:
        print(row)
    # return the last element of the matrix
    return K[n][w]
```

In [ ]:

```
import unittest

class TestZeroOneKnapsack(unittest.TestCase):

    def test_small_values(self):
        W = 5
        wt = [2, 3, 4]
        vals = [3, 4, 5]
        expected_output = 7
        print("\nTest 1 - Small Values")
        self.assertEqual(Zero_One_Knapsack(len(vals), wt, vals, W), expected_out

    def test_repeated_weights(self):
        W = 7
        wt = [2, 3, 3, 4]
        vals = [3, 4, 5, 6]
        expected_output = 11
        print("\nTest 2 - Repeated Weights")
        self.assertEqual(Zero_One_Knapsack(len(vals), wt, vals, W), expected_out

    def test_same_weight(self):
        W = 10
        wt = [5, 5, 5, 5]
        vals = [10, 20, 30, 40]
        expected_output = 70
        print("\nTest 3 - Same Weight")
        self.assertEqual(Zero_One_Knapsack(len(vals), wt, vals, W), expected_out

    def test_same_value(self):
        W = 8
        wt = [2, 3, 4, 5]
        vals = [5, 5, 5, 5]
        expected_output = 10
        print("\nTest 4 - Same Value")
        self.assertEqual(Zero_One_Knapsack(len(vals), wt, vals, W), expected_out

    def test_single_item(self):
        W = 3
        wt = [2]
        vals = [4]
        expected_output = 4
        print("\nTest 5 - Single Item")
        self.assertEqual(Zero_One_Knapsack(len(vals), wt, vals, W), expected_out

# Create a test suite
suite = unittest.TestLoader().loadTestsFromTestCase(TestZeroOneKnapsack)

# Run the test suite and print the results
runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

```
test_repeated_weights (__main__.TestZeroOneKnapsack.test_repeated_weights) ... ok
test_same_value (__main__.TestZeroOneKnapsack.test_same_value) ... ok
test_same_weight (__main__.TestZeroOneKnapsack.test_same_weight) ... ok
test_single_item (__main__.TestZeroOneKnapsack.test_single_item) ... ok
test_small_values (__main__.TestZeroOneKnapsack.test_small_values) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.004s

OK
Test 2 - Repeated Weights
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 3, 3, 3, 3, 3, 3]
[0, 0, 3, 4, 4, 7, 7, 7]
[0, 0, 3, 5, 5, 8, 9, 9]
[0, 0, 3, 5, 6, 8, 9, 11]

Test 4 - Same Value
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 5, 5, 5, 5, 5, 5, 5]
[0, 0, 5, 5, 5, 10, 10, 10, 10]
[0, 0, 5, 5, 5, 10, 10, 10, 10]
[0, 0, 5, 5, 5, 10, 10, 10, 10]

Test 3 - Same Weight
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 10, 10, 10, 10, 10, 10]
[0, 0, 0, 0, 0, 20, 20, 20, 20, 20, 30]
[0, 0, 0, 0, 0, 30, 30, 30, 30, 30, 50]
[0, 0, 0, 0, 0, 40, 40, 40, 40, 40, 70]

Test 5 - Single Item
[0, 0, 0, 0]
[0, 0, 4, 4]

Test 1 - Small Values
[0, 0, 0, 0, 0, 0]
[0, 0, 3, 3, 3, 3]
[0, 0, 3, 4, 4, 7]
[0, 0, 3, 4, 5, 7]
```

Out[ ]: `<unittest.runner.TextTestResult run=5 errors=0 failures=0>`

Different style of testing

In [ ]:
```python
def test_Zero_One_Knapsack():
    test_cases = [
        ((5, [2, 3, 4], [3, 4, 5]), 7),
        ((7, [2, 3, 3, 4], [3, 4, 5, 6]), 11),
        ((10, [5, 5, 5, 5], [10, 20, 30, 40]), 70),
        ((8, [2, 3, 4, 5], [5, 5, 5, 5]), 10),
        ((3, [2], [4]), 4)
    ]

    for i, ((W, wt, vals), expected) in enumerate(test_cases):
        result = Zero_One_Knapsack(len(vals), wt, vals, W)
        assert result == expected, f"Test case {i} failed: expected {expected},
        print(f"Test case {i} passed\n")
```

```python
test_Zero_One_Knapsack()
```

```
[0, 0, 0, 0, 0, 0]
[0, 0, 3, 3, 3, 3]
[0, 0, 3, 4, 4, 7]
[0, 0, 3, 4, 5, 7]
Test case 0 passed

[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 3, 3, 3, 3, 3, 3]
[0, 0, 3, 4, 4, 7, 7, 7]
[0, 0, 3, 5, 5, 8, 9, 9]
[0, 0, 3, 5, 6, 8, 9, 11]
Test case 1 passed

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 10, 10, 10, 10, 10, 10]
[0, 0, 0, 0, 0, 20, 20, 20, 20, 20, 30]
[0, 0, 0, 0, 0, 30, 30, 30, 30, 30, 50]
[0, 0, 0, 0, 0, 40, 40, 40, 40, 40, 70]
Test case 2 passed

[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 5, 5, 5, 5, 5, 5, 5]
[0, 0, 5, 5, 5, 10, 10, 10, 10]
[0, 0, 5, 5, 5, 10, 10, 10, 10]
[0, 0, 5, 5, 5, 10, 10, 10, 10]
Test case 3 passed

[0, 0, 0, 0]
[0, 0, 4, 4]
Test case 4 passed
```

Ask Kiran about pros and cons for each testing style