Top-Down Approach with Memoization

Whenever we solve a subproblem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. This technique of storing the results of the previously solved subproblems is called Memoization.

In memoization, we solve the bigger problem by recursively finding the solution to the subproblems. It is a top-down approach.

```
#Implementation of Fibonacci Series using Memoization
""" The Fibonacci Series is a series of numbers in which each number
    is the sum of the preceding two numbers.
    By definition, the first two numbers are 0 and 1.

    Implement with the following steps:
    - Declare function with parameters: Number N and Dictionary Memo.
    - If n equals 1, return 0
    - If n equals 2, return 1
    - If current element is not in memo, add to memoy by recursive call for previous functi
```

```
#Implementation of Factorial of a number N using Memoization
def factorial(n):
  if n == 1:
    return 1

  else:
    return(n * factorial(n-1))

n = 10
d = factorial(n)
print('The factorial of',n, 'is:', d)
```

```
 The factorial of 10 is: 3628800
```

Bottom-Up Approach with Tabulation

Tabulation is the opposite of the top-down approach and does not involve recursion. In this approach, we solve the problem "bottom-up". This means that the subproblems are solved first and are then combined to form the solution to the original problem.

This is achieved by filling up a table. Based on the results of the table, the solution to the original problem is computed.

```
#Implementation of Fibonacci Series using Tabulation
```

```
#Implementation of Fibonacci Series using Tabulation
""" Fibonacci Series can be implemented using Tabulation using the following steps:
    - Declare the function and take the number whose Fibonacci Series is to be printed.
    - Initialize the list and input the values 0 and 1 in it.
    - Iterate over the range of 2 to n+1.
    - Append the list with the sum of the previous two values of the list.
    - Return the list as output. """

#Implementation of Factorial of a number N using Tabulation
```

Answers

```
#Lecture Scheduling Problem
""" You are given a set of N schedules.
    Schedule will be defined with: course code, units, start, duration.
    Select the maximum set of lectures to be held.
    No schedules must overlap. """

class Lecture(object):
  def __init__(self, c, u, s, d):
    self.courseCode = c
    self.units = u
    self.startTime = s
    self.duration = d

  def getUnits(self):
    return self.units

  def getDuration(self):
    return self.duration

  def getStart(self):
    return self.startTime

  def density(self):
    return self.getUnits()/self.getDuration()

  def __str__(self):
    return self.courseCode + ': <' + str(self.units) + ', ' + str(self.startTime) + ', '

def buildSched(codes, units, start, duration):
  schedule = []
  for i in range(len(start)):
    schedule.append(Lecture(codes[i], units[i], start[i], duration[i]))
  return schedule

def greedy(classes, maxHours, keyFunction):
  classesCopy = sorted(classes, key = keyFunction, reverse = False)
```

```
  result = []

  totalUnits, totalHours = 0.0, 0.0
  for i in range(len(classes)):
    if(totalHours + classesCopy[i].getDuration()) <= maxHours:
      result.append(classesCopy[i])
      totalHours += classesCopy[i].getDuration()
      totalUnits += classesCopy[i].getUnits()
  return (result, totalUnits)

def testGreedy(classes, constraint, keyFunction):
  taken, val = greedy(classes, constraint, keyFunction)
  print('Total units of classes taken = ', val)
  for item in taken:
    print(' ', item)

def testGreedys(ClassCodes, maxHours):
    print('Use greedy by value to allocate', maxHours,          'units')
    testGreedy(ClassCodes, maxHours, Lecture.getUnits)
    print('\nUse greedy by cost to allocate', maxHours,          'units')
    testGreedy(ClassCodes, maxHours, Lecture.getDuration)
    print('\nUse greedy by density to allocate', maxHours,          'units')
    testGreedy(ClassCodes, maxHours, Lecture.density)

courseCode = ['CPE001', 'CPE002', 'CPE003']
units = [2, 3, 4]
startTime = [9, 2, 3]
Duration = [2, 2, 3]
enrolledClasses = buildSched(courseCode, units, startTime, Duration)
testGreedys(enrolledClasses, 5)
```

```
Use greedy by value to allocate 5 units
Total units of classes taken =  5.0
  CPE001: <2, 9, 2>
  CPE002: <3, 2, 2>

Use greedy by cost to allocate 5 units
Total units of classes taken =  5.0
  CPE001: <2, 9, 2>
  CPE002: <3, 2, 2>

Use greedy by density to allocate 5 units
Total units of classes taken =  6.0
  CPE001: <2, 9, 2>
  CPE003: <4, 3, 3>
```

```
#Student Enrollment Problem
""" You are given a set of N courses.
    Courses will be defined as (code, units).
    Select the maximum set of courses a student can hve.
    Maximum units allowed is based on input with a max of 24
    default of 18. """
```

```
    default of 18. """
```

```
# Function to implement Fibonacci Series
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)

#Printing the elements of the Fibonacci Series
print("0")
print("1")
for element in tempDict.values():
    print(element)

# Function to implement Fibonacci Series
def fibTab(n):
    tb = [0, 1]
    for i in range(2, n+1):
      tb.append(tb[i-1] + tb[i-2])
    return tb

print(fibTab(6))
```