Greedy Algorithm

Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

When to Use Greedy Algorithms

Greedy Algorithms can help you find solutions to a lot of seemingly tough problems. The only problem with them is that you might come up with the correct solution but you might not be able to verify if its the correct one. All the greedy problems share a common property that a local optima can eventually lead to a global minima without reconsidering the set of choices already considered.

Sample Problems:

1. Lecture Scheduling Problem
2. Student Enrollment Problem

Brute Force Algorithm

Brute Force Algorithms are exactly what they sound like – straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency. For example, imagine you have a small padlock with 4 digits, each from 0-9.

Sample Problems:

- Solve the same given problems using Brute Force algorithm.

Dynamic Programming

Dynamic Programming(DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to the subproblems.

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 function and add. """

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]

MemoDict = {}
FIBMemo(10, MemoDict)

print("0")
```

```
print("1")
for element in MemoDict.values():
    print(element)

def fact(n):
  if n == 1:
      return 0
  if n == 2:
      return 1
  if n not in memo:
    memo[n] = n * fact(n - 1)
    return memo[n]

memo = {}

for i in range(1, 10):
  print("Factorial")
  print(fact(i))
```

```
0
1
1
2
3
5
8
13
21
34
Factorial
0
Factorial
1
Factorial
3
Factorial
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-b74d2fb5cff1> in <cell line: 40>()
     40 for i in range(1, 10):
     41   print("Factorial")
---> 42   print(fact(i))

<ipython-input-38-b74d2fb5cff1> in fact(n)
     33       return 1
     34   if n not in memo:
---> 35     memo[n] = n * fact(n - 1)
     36     return memo[n]
     37

TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

SEARCH STACK OVERFLOW

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
""" 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. """

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

print(FIBTb(10))
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

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) + ', ' + str(self.duration) + '>'


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. """
```

```
# 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)
```

```
     0
     1
     1
     2
     3
     5
```

```
# 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))
```

```
     [0, 1, 1, 2, 3, 5, 8]
```