```python
# -*- coding: utf-8 -*-
"""
Created on Fri Jun  8 01:01:46 2018

@author: Tawanda Vera
"""
from __future__ import print_function
from __future__ import division
from __future__ import unicode_literals
import sys
import collections
import copy
from functools import wraps

# A Memoizing Decorator to speed up the algoritnm running times
def memo(func):
    cache = {}                           # Stored subproblem solutions
    @wraps(func)                         # Make wrap look like func
    def wrap(*args):                     # The memoized wrapper
        if args not in cache:            # Not already computed?
            cache[args] = func(*args)    # Compute & cache the solution
            return cache[args]            # Return the cached solution
        return wrap                          # Return the wrapper


##############################################################################
# Problem 1:   Rod-Cutting problem with more constraints
#
# Consider a modification of the rod-cutting problem in which, in addition to
# a price pi for each rod, each cut incurs a fixed cost of c. The revenue
# associated with a solution is now the sum of the prices of the pieces minus
# the costs of making the cuts. Give a dynamic-programming algorithm to solve
# this modified problem.
##############################################################################

# price for each rod using DP

price={1:1,2:5,3:8,4:9,5:10,6:17,7:17,8:20,9:24,10:30}
max_price = {}

def cut_rod(n,price):
    if n==0:
        # We are returning 0 and the empty list for a rod of size 0 as we
        # do not cut. This is our base case
        return 0,[]
    revenue=-sys.maxsize - 1

    # If we have already calculated the max price for the length, return price
    # and indicies
    if n in max_price.keys():
        return max_price[n][0],copy.copy(max_price[n][1])
    cuts = []
    for i in range(1,n+1):
            #Get the revenue and indicies from the smaller piece
        smaller_revenue,smaller_cuts =  cut_rod(n-i,price)
            #If we get a better price, set the indicies and revenue for the length
        if smaller_revenue + price[i] > revenue:
```

```
            smaller_cuts.append(i)
            cuts = smaller_cuts
            revenue = smaller_revenue + price[i]
            cost = 1    # Assume a fixed cost of making cut of 1
        if len(cuts)> 1:
            revenue = revenue - cost  # fixed cost of making cut
            cuts = cuts
        else:
            revenue = revenue
    #store the calculated max results for rod of length n
    #need to copy to avoid linking
    max_price[n] = (revenue,copy.copy(cuts))
    return revenue, cuts


# modification of the rod-cutting problem
"""The major modification was in the fixed cost of
making the cut, which is deducted from the revenue. The program also handles
 the case in which we make no cuts (len(cut)= 1); The total revenue in this
 case is simply 'revenue'. In the case of cuts, we would be deducting cost
 from the total revenue to give revnue[n]
"""
# No cuts
n = 3
revenue, cuts = cut_rod(n,price)
print("revenue for rod of length %d is %d" %(n,revenue)),
print("Cuts were sizes:", cuts),
print("key = rod length,value = (revenue, [where cuts were made]:", max_price)
# revenue for rod of length 3 is 8
# Cuts were sizes: [3]
# key = rod length,
# value = (revenue, [where cuts were made]:
# {1: (1, [1]), 2: (5, [2]), 3: (8, [3])}

# With cuts
n = 8
revenue, cuts = cut_rod(n,price)
print("revenue for rod of length %d is %d" %(n,revenue)),
print("Cuts were sizes:", cuts),
print("key = rod length,value = (revenue, [where cuts were made]:", max_price)

# revenue for rod of length 8 is 19
# Cuts were sizes: [2, 6]
# key = rod length,
# value = (revenue, [where cuts were made]:
# {1: (1, [1]), 2: (5, [2]), 3: (8, [3]), 4: (9, [4]),
# 5: (10, [2, 3]), 6: (17, [6]), 7: (16, [1, 6]), 8: (19, [2, 6])}

################################################################################
# Problem 2:   Longest palindrome subsequence

# A palindrome is a nonempty string over some alphabet that reads the same
# forward and backward. Examples of palindromes are all strings of length 1,
# civic, racecar, and aibohphobia (fear of palindromes).

# Give an efficient algorithm to find the longest palindrome that is a
# subsequence of a given input string. For example, given the input character,
```

```
# your algorithm should return carac.

# What is the running time of your algorithm?

################################################################################

# LPS problem
# Returns the length of the longest palindromic subsequence in seq

# Step 1: Reverse the given sequence and store the reverse in another array
# Function to reverse a string
def reverse(string):
    string = "".join(reversed(string))
    return string

# The Longest Common Subsequence (LCS) of the given sequence and rev[] will be
# the longest palindromic sequence.


# Step 2: Create the Longest Common Subsequence (LCS) Function
# Returns length of LCS for X[0..m-1], Y[0..n-1]
def lcs(X, Y, m, n):
   L = [[0 for x in range(n+1)] for x in range(m+1)]

   # Following steps build L[m+1][n+1] in bottom up fashion. Note
   # that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1]
   for i in range(m+1):
      for j in range(n+1):
         if i == 0 or j == 0:
            L[i][j] = 0
         elif X[i-1] == Y[j-1]:
            L[i][j] = L[i-1][j-1] + 1
         else:
            L[i][j] = max(L[i-1][j], L[i][j-1])

   # Following code is used to print LCS
   index = L[m][n]

   # Create a character array to store the lcs string
   lcs = [""] * (index+1)
   lcs[index] = ""

   # Start from the right-most-bottom-most corner and
   # one by one store characters in lcs[]
   i = m
   j = n
   while i > 0 and j > 0:

      # If current character in X[] and Y are same, then
      # current character is part of LCS
      if X[i-1] == Y[j-1]:
         lcs[index-1] = X[i-1]
         i-=1
         j-=1
         index-=1

      # If not same, then find the larger of two and
```

```
        # go in the direction of larger value
        elif L[i-1][j] > L[i][j-1]:
            i-=1
        else:
            j-=1

    print ("LPS of " + X + " is " + "".join(lcs))


# Testing Code
X = "character"
Y = reverse(X)
m = len(X)
n = len(Y)
memo(lcs(X, Y, m, n))

# LPS of character is carac

# The running time of the algorithm = O(m*n)  = O(n**2)
#T(n) = O(n*m)
"""The running time of  the algorithm is a O(n^2) """

################################################################################

# Problem 3:   Fixing Corrupted Text Documents

# You are given a string of n characters s[1. . . n], which you believe to be
# a corrupted text document in which all punctuation has vanished (so that it
# looks something like “itwasthebestoftimes...” ). You wish to reconstruct the
# document using a dictionary, which is available in the form of a Boolean
# function dict(.) : for any string w,

# 1. Give a dynamic programming algorithm that determines whether the string
# s[.] can be reconstituted as a sequence of valid words.
# The running time should be at most O(n2). Implement your algorithm in python.

# 2.Make your algorithm output the corresponding sequence of word. Use a
# choice array to record the optimal choices made by your algorithm and then
# use a backtracking algorithm to output the sequence of words.

################################################################################

def wordBreak(s, wordDict):
        """
        :type s: str
        :type wordDict: Set[str]
        :rtype: List[str]
        """
        dic = collections.defaultdict(list)

        def breakstring(s):
            n = len(s)
            dp = [False] * (n + 1)
            dp[0] = True
            for i in range(n):
                for j in range(i, -1, -1):
                    if dp[j] and s[j:i + 1] in wordDict:
```

```
                    dp[i + 1] = True
                    break
        return dp[n]

    def validword(s):
        if not s:
            return [None]
        if breakstring(s)== True:
            return dic[s]
        res = []
        # In the event that the string is valid,
        # Use a choice array to record the optimal
        # choices made by your algorithm and
        # then use a backtracking algorithm to output
        # the sequence of words.
        for word in wordDict:
            n = len(word)
            if s[:n] == word:
                for r in validword(s[n:]):
                    if r:
                        res.append(word + " " + r)
                    else:
                        res.append(word)
        dic[s] = res
        return res
    return validword(s)

# Testing code
s ='itwasthebestoftimes'
wordDict = {"it", "was", "the", "best", "of", "times"}
print("The WordBreak of ", [ s ], " is ", wordBreak(s,wordDict))

# The WordBreak of  ['itwasthebestoftimes']  is  ['it was the best of times']

################################################################################

# Application of Dynamic Programming Algorithms in Quantitative Finance

################################################################################
"""
Alpha generation models are used to locate excess return in the capital market.
 They enable the development of mathematical and statistical models that help
 determine whether or not a specific investment may be profitable. As Quants,
 our aim is to develop trading systems that optimizes some profit criterion,
 the simplest being the total return. Given a set of instruments, a trading
 strategy is a switching function that transfers the wealth from one instrument
 to another. For instance, finding the optimal trading strategy for non-zero
 transactions cost is a path dependent optimization problem even when the price
 time series is known. A brute force approach to solving this problem would
 search through the space of all possible trading strategies, keeping only the
 one satisfying the optimality criterion. Since the number of possible trading
 strategies grows exponentially with time, the brute force approach leads to
 an exponential time algorithm, which for all practical purposes is infeasible
 – even given the pace at which computing power grows.

The dynamic programming (DP) methods used in this assignment include the
memoization, longest common sequence, longest palindrome sequence, and
```

```
backtracking. The DP methods used in this assignment are useful in alpha
generation models as they can breakdown the problem in sub problems and sub-
sub problems, thus reducing the time complexity and providing optimal solutions
 to decision problems. To be more specific, note that the optimal strategy
 constructed by DP algorithms requires full knowledge of the future price
 paths. The DP approach can be used to generate sample paths for the
 instruments. These sample paths can be used to compute the optimal trading
 strategy given the current history and information set. One then has a sample
 set of future paths and corresponding optimal trading strategies on which to
 base the current action. Suppose that our alpha model signals us that it is
 profitable to liquidate a large number N of stock at price St and we wish to
 do so by the end of the day at time T. Realistically, market does not have
 infinite liquidity, so it cannot absorb a large sell order at the best
 available price, which means we will walk the order book or even move the
 market and execute an order at a lower price (subject to market impact
 denoted as ‘h’). Hence, we should spread this out over time, and solve a
 stochastic control problem. We may also have a sense of urgency, represented
 by penalising utility function for holding non-zero inventory throughout the
 strategy.
 """
 ###############################################################################
```