```
# -*- coding: utf-8 -*-
"""
Created on Mon Jun 11 03:59:14 2018

@author: Tawanda Vera
"""

from __future__ import print_function
from __future__ import division
from __future__ import unicode_literals
from functools import wraps
import string

# 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:  Huffman code for Fibonacci numbers

# What is an optimal Huffman code for the following set of frequencies, based
# on the first 8 Fibonacci numbers: a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

#Can you generalize your answer to find the optimal code when the frequencies
# are the first n Fibonacci numbers?

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

# Huffman Coding

def huffman(p):
    '''Return a Huffman code for an ensemble with distribution p.'''
    # Base case of only two symbols, assign 0 or 1 arbitrarily
    if len(p) == 2:
        return dict(zip(p.keys(), ['0', '1']))

    # Create a new distribution by merging lowest prob. pair
    p_prime = p.copy()
    a1, a2 = lowest_prob_pair(p)
    p1, p2 = p_prime.pop(a1), p_prime.pop(a2)
    newkey = (a1, a2)
    p_prime[newkey] = p1 + p2

    # Recurse and construct code on new distribution
    c = huffman(p_prime)
    ca1a2 = c.pop(newkey)
    c[a1] = ca1a2 + '0'
    c[a2] = ca1a2 + '1'
    return c

def lowest_prob_pair(p):
    """Return pair of symbols from distribution p with lowest probabilities."""
    assert len(p) >= 2 # Ensure there are at least 2 symbols in the dist.
    sorted_p = sorted(p.items(),key=lambda kv:kv[1])
    return sorted_p[0][0], sorted_p[1][0]
```

```
# 8 items from the Fibonnacci Sequence
fib = {'a':1, 'b':1, 'c':2, 'd':3, 'e':5,'f':8, 'g':13,
            'h':21}
#Testing Code
huff = huffman(fib)

# Printing output
print ("Symbol".ljust(10) + "Weight".ljust(10) + "Huffman Code")

for value in huff:
    print(value[:1].ljust(10) + str(fib[value[0]]).ljust(10) + str(huff[value[:1]]))

#output
"""
Symbol    Weight    Huffman Code
h         21        0
g         13        10
f         8         110
e         5         1110
d         3         11110
c         2         111110
a         1         1111110
b         1         1111111
"""

# Can you generalize your answer to find the optimal code when the frequencies
# are the first n Fibonacci numbers?

def fib_recursive(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recursive(n-1) + fib_recursive(n-2)

def fib_dict(n):
    fib = [fib_recursive(i) for i in range(n)]
# create Fib dictionary with alphabet
    if len(fib) == n:
        fib_dict = dict(zip(string.ascii_letters, fib))
    return fib_dict

def FibHuff(n):
    huff= huffman(fib_dict(n))
    fibDict = fib_dict(n)
    # Printing output
    print ("Symbol".ljust(10) + "Weight".ljust(10) + "Huffman Code")
    for value in huff:
        print(value[:1].ljust(10) + str(fibDict[value[0]]).ljust(10) + str(huff[value[:1]]))


# use memoization to speed the processing time for very large n
memo(FibHuff(32))

# n = 32 was the highest nth Fibonnacci Sequence my computer could run

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

# Problem 2:   Coin Changing

# Consider the problem of making change for n cents using the fewest number of
```

```
# coins. Assume that each coin's value is an integer. Describe a greedy
# algorithm to make change consisting of quarters, dimes, nickels, and pennies.

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

#coins
quarter = 25
dime = 10
nickle = 5
penny = 1

#Assumed quantities
qnty = {quarter:100, dime:100, nickle:100, penny:100}

#Basic change
def default_change():
    return {quarter:0, dime:0, nickle:0,penny:0}

# Target change
def total(money):
    r = 0
    for coin in money:
        r += money[coin]
    return r

# Greedy Coin Change
def make_change(to_break):
    if total(qnty) < to_break:
        return {}
    r = default_change()
    for coin in qnty:
        while qnty[coin] > 0 and coin <= to_break:
            qnty[coin] -= 1
            r[coin] += 1
            to_break -= coin
    return r

# Testing Code on (100,200, ....24 cents)
for breaking in [100, 200, 201, 0, 1, 6, 11,24]:
    print (" ".ljust(25) + "quarter".ljust(10),
           "dime".ljust(10) + "nickle".ljust(10) + "penny:")
    print("Amount of coins: {}".format(total(qnty)) )
    print ("Available denominations: {}".format(qnty))

    print("Breaking ${}".format(breaking))
    print("Change made: {}".format(make_change(breaking)))

################################################################################
```