```
#--------------------------------------------------------------------------------
# Author:       Alex Banning
# Assignment:   Programming Assignment 1: RSA Cracker
# Date:         September 19th, 2019
# Filename:     RSACracker.py
# Description:  Runs a brute force attack to decrypt RSA-encrypted ciphertext
#               when given a public key and original prime number.
#               This program claculates all the intermediate steps modularly for
#               ease of use.
#--------------------------------------------------------------------------------

# Description: Encrypts a plaintext integer using the public key pair {e, n}
# pre:         P is a plaintext integer, e and n contain values calculated in
#              the RSA encryption process.
# post:        Returns the ciphertext of the message after encrypting
# usage:       rsaEncrypt(15, 7, 33)
#--------------------------------------------------------------------------------
def rsaEncrypt(P, e, n):
    C = (P ** e) % n
    return C
#--------------------------------------------------------------------------------

# Description: Decrypts a ciphertext integer using the private key pair {d, n}
# pre:         C is a ciphertext integer, d and n contain the private key pair
#              for decryption
# post:        Returns the plaintext of the encrypted message after decrypting
# usage:       rsaDecrypt(14, 3, 33)
#--------------------------------------------------------------------------------
def rsaDecrypt(C, d, n):
    P = (C ** d) % n
    return P
#--------------------------------------------------------------------------------

# Description: Cracks the RSA encryption by figuring out the private key 'd'
#              using a brute force attack
# pre:         e and n contain the values of the public key pair {e, n}
# post:        returns the private key 'd'
# usage:       crackRSA(7, 33)
#--------------------------------------------------------------------------------
def crackRSA(e, n):
    pqArray = findPrimeFactors(n)
    phiN = calculatePhiN(pqArray[0], pqArray[1])
    d = getPrivateKey(e, phiN)
    return d
#--------------------------------------------------------------------------------

# Description: crackRSA helper function for modularity
# pre:         e contains the public key, and phiN contains the Euler totient
#              of the prime factors of n
# post:        returns the private key 'd'
# usage:       getPrivateKey(e, phiN)
#--------------------------------------------------------------------------------
def getPrivateKey(e, phiN):
    for d in range(1, phiN):
        if isGoodD(d, e, phiN):
            return d
    return 0
#--------------------------------------------------------------------------------

# Description: Checks to see if a number is prime
# pre:         num contains an integer
# post:        Returns true if num is prime, false otherwise
# usage:       if isPrime(num):
#--------------------------------------------------------------------------------
def isPrime(num):
    if (num == 1) | (num == 0):
        return False
    i = 3
    while ((i * i) <= num):
        if ((num % i == 0) & (i != 2)):
            return False
        i += 1
    return True
#--------------------------------------------------------------------------------

# Description: Calculates the Euler totient of the inputs
# pre:         p and q are both prime numbers and p != q
# post:        returns the Euler totient of the values
# usage:       phiN = calculatePhiN(3, 11)
#--------------------------------------------------------------------------------
def calculatePhiN(p, q):
    phiN = (p - 1) * (q - 1)
    return phiN
#--------------------------------------------------------------------------------

# Description: Checks to see if the GCD of two values is 1
# pre:         num1 and num2 contain integer values
# post:        Returns true if the gcd of num1 and num2 is 1, false otherwise
# usage:       if isGoodGCD(i, phiN):
#--------------------------------------------------------------------------------
def isGoodGCD(num1, num2):
    if gcd(num1, num2) == 1:
        return True
    else:
        return False
#--------------------------------------------------------------------------------

# Description: Returns the gcd of two values
# pre:         a and b contain integer values
# post:        Returns gcd(a, b)
# usage:       theGCD = gcd(64, 48)
#--------------------------------------------------------------------------------
def gcd(a, b):
    if (a > b):
        value = gcdHelper(a, b)
    else:
        value = gcdHelper(b, a)
    return value
#--------------------------------------------------------------------------------

# Description: Recursive GCD Helper function for modularity. Implements
#              Euclidean algorithm
# pre:         a and b contain integer values
# post:        Returns the gcd of the two values
# usage:       value = gcdHelper(a, b)
#--------------------------------------------------------------------------------
def gcdHelper(a, b):
    if b == 0:
        return a
    else:
        bNext = a % b
        return gcdHelper(b, bNext)
#--------------------------------------------------------------------------------

# Description: Checks to see if a 'd' value is good or not
# pre:         d contains the 'd' to be tested, e contains the public key,
#              and phiN is the Euler totient of the factors of n
# post:        Returns true if de % phiN == 1, false otherwise
# usage:       if (isGoodD(d, e, phiN)):
#--------------------------------------------------------------------------------
def isGoodD(d, e, phiN):
    if (d * e) % phiN == 1:
        return True
    else:
        return False
#--------------------------------------------------------------------------------

# Description: Selects an 'e' value according to the RSA Encryption steps
# pre:         phiN is the Euler totient of the prime factors of key n
# post:        Returns an 'e' value that fits the algorithm's cases
# usage:       e = selectE(phiN)
#-------------------------------------------------------------------------------
def selectE(phiN):
    for i in range(1, phiN):
        if isGoodGCD(i, phiN):
            e = i
        else:
            e = 0
    return e
#-------------------------------------------------------------------------------

# Description: Finds the prime factors of a given number
# pre:         n contains a value
# post:        Returns an array of the prime factors of n
# usage:       factors = []; factors = findPrimeFactors(n);
#-------------------------------------------------------------------------------
def findPrimeFactors(n):
    factors = []
    hasNotBeenFound = True
    i = 3
    while ((hasNotBeenFound) & ((i * i) <= n)):
        if n % i == 0 & isPrime(i):
            factors.append(i)
            hasNotBeenFound = False
            i += 1
        else:
            i += 1
    temp = (n / factors[0])
    factors.append(temp)
    return factors
#-------------------------------------------------------------------------------

# Description: Test client for testing decryption
# pre:         None
# post:        Prints output from all test cases with given variables
# usage:       testRSACracker()
#-------------------------------------------------------------------------------
def testRSACracker():
    testCaseNoArray = [1, 2, 3, 4, 5]
    nArray = [33, 55, 77, 143, 527]
    eArray = [7, 3, 17, 11, 7]
    dExpectedArray = [3, 27, 53, 11, 343]
    plaintextArray = [5, 9, 8, 7, 2]
    ciphertextExpectedArray = [14, 14, 57, 106, 128]

    for i in range(0, 5):
        testCase(testCaseNoArray[i], nArray[i], eArray[i], dExpectedArray[i], \
        plaintextArray[i], ciphertextExpectedArray[i])

    print("************************************************************")
    print("Testing concluded for all given cases.")
    print("")

    return 0
#-------------------------------------------------------------------------------

# Description: testRSACracker helper function for modularity
# pre:         testCaseNo, n, e, dExpected, plaintext, and
#              ciphertextExpectedArray are all given values from top-level test
#              client.
# post:        Prints outputs to the screen
# usage:       for i in range(0, 5):
#                  testCase(testCaseNoArray[i]....);
#-------------------------------------------------------------------------------
def testCase(testCaseNo, n, e, dExpected, plaintext, ciphertextExpected):
    # For Test Case X -------------------------------------------------------
    # Calculate Necessary Values
    d = crackRSA(e, n)
    ciphertext = rsaEncrypt(plaintext, e, n)
    plaintextDecrypted = rsaDecrypt(ciphertext, d, n)

    # Print Results
    print("*********************** Test Case "),
    print(testCaseNo),
    print("***********************")

    print("d (expected - actual):           "),
    print(dExpected),
    print("  "),
    print(d)

    print("ciphertext (expected - actual):  "),
    print(ciphertextExpected),
    print("  "),
    print(ciphertext)

    print("plaintext (expected - actual):   "),
    print(plaintext),
    print("  "),
    print(plaintextDecrypted)
    print(""),
    print("")
    #----------------------------------------------------------------------------
#--------------------------------------------------------------------------------

# Running code
testRSACracker()
```