

PSEUDOPRIMES

PROJECT REPORT

Submitted for CAL in B.Tech Data Structures and Algorithms (CSE2003)

By

Meghna Lohani (16BLC1103)

Sruthi Shiva (16BCE1064)

Piyush Singh (16BEC1120)

Iona Thomas (16BCE1368)

Slot: B1

Name of the Faculty: Dr. Nagaraj S.V.

(SCHOOL OF COMPUTER SCIENCE AND ENGINEERING)



April, 2017

ABSTRACT

Pomerance, Selfridge and Wagstaff offered \$30 for a number n which is simultaneously a strong base 2-pseudoprime and a Lucas pseudoprime (with a discriminant specified). Pomerance, Selfridge and Wagstaff showed that there are no counterexamples less than $20 * 10^9$. With time, the prize for such a number has grown to \$620, and the conditions have been relaxed. In this project, we study algorithms for tabulating pseudoprimes including special ones such as those mentioned.

Introduction

A pseudoprime is a probable prime (an integer that shares a property common to all prime numbers) that is not actually prime. Pseudoprimes are classified according to which property of primes they satisfy.

The Baillie-PSW (BPSW or BSW) primality test is a compositeness test, in the manner of Fermat's test and the Miller-Rabin test. It is named for Robert Baillie, Carl Pomerance, John L. Selfridge, and Samuel S. Wagstaff, Jr. The algorithm was apparently first conceived by Baillie (Baillie and Wagstaff, 1980), with refinements added by Selfridge (Pomerance, Selfridge, and Wagstaff, 1980).

The Baillie – PSW primality test is a probabilistic primality testing algorithm that determines whether a number is composite or is a probable prime. It is a combination of a strong Fermat probable prime test to base 2 and a strong Lucas probable prime test. The Fermat and Lucas test each has its own list of pseudoprimes, that is, composite numbers that pass the primality test.

The power of the Baillie-PSW test comes from the fact that these lists of strong Fermat pseudoprimes and strong Lucas pseudoprimes have no known overlap. There is even evidence that the numbers in these lists tend to be different kinds of numbers.

Pseudoprimes base 2 tend to fall into the residue class 1 (mod m) for many small m , whereas Lucas pseudoprimes tend to fall into the residue class -1 (mod m). As a result, a number that passes both a strong Fermat and a strong Lucas test is very likely to be prime.

No composite number below 264 passes the Baillie-PSW test. Consequently, this can be considered a deterministic primality test on numbers below that bound. There are also no known composite numbers above that bound that pass the test.

As of this date, both the standard and strong tests are flawless; no counterexample (BPSW standard or strong pseudoprime) is known. With the aid of Richard G. E. Pinch's table of base-2 strong pseudoprimes, the author verified (May, 2005) that no BPSW pseudoprime (standard or strong) exists for $N < 10^{13}$. In January, 2007, with the aid of the author's code and William Galway's table of pseudoprimes, Martin Fuller determined that no BPSW pseudoprime (standard or strong) exists for $N < 10^{15}$. More recently, Jeff Gilchrist, using the author's code and a database of pseudoprimes prepared by Jan Feitsma, has verified (13 June 2009) that no BPSW pseudoprime (standard or strong) exists below 10^{17} .

Furthermore, analysis (24 October 2009) by Gilchrist of an extension of Feitsma's database to 2^{64} found no BPSW pseudoprime (standard or strong) below 2^{64} (approximately $1.8446744e19$). This lower bound of 2^{64} for any BPSW pseudoprime has since been separately verified (11 July 2011) by Charles Greathouse; however, he also used Feitsma's database, so a completely independent check has not yet been carried out.

Note, however, that Carl Pomerance (1984) presented a heuristic argument that an infinite number of counterexamples exist to the standard test (and presumably to the strong test as well, based on

similar reasoning), and even that (for sufficiently large x , dependent on μ) the number of BPSW pseudoprimes $< x$ exceeds $x^{(1-\mu)}$, where μ is an arbitrarily small pre-assigned positive number. Nevertheless, not a single BPSW pseudoprime has yet been found. Consequently, the BPSW test carries an aura of dependability (justified or not) exceeding that of competing algorithms, such as multiple Miller-Rabin tests.

It is believed that some commercial mathematics software packages rely, in part or in whole, on the BPSW test for primality checking; see, for example, Ribenboim (1995/6, p. 142), also Martin (2004). In some instances, these packages appear to use the strong BPSW and/or Lucas test, or add additional Miller-Rabin tests.

BPSW requires $O((\log n)^3)$ bit operations, as do Fermat's test and the Miller-Rabin algorithm. However, a BPSW test typically requires roughly three to seven times as many bit operations as a single Miller-Rabin test. The strong version of BPSW differs only in replacing the standard Lucas-Selfridge test with the strong Lucas-Selfridge test. The strong Lucas-Selfridge test produces only roughly 30 % as many pseudoprimes as the standard version; for example, among the odd composites $N < 10^6$, there are 219 standard Lucas-Selfridge pseudoprimes, 58 strong Lucas-Selfridge pseudoprimes, and 46 base-2 strong pseudoprimes (these totals presume no screening with odd trial divisors). Since the strong Lucas-Selfridge test incurs roughly 10 % more running time, the strong tests appear to be more effective.

Selfridge specified the following parameters for the generation of the Lucas sequences: $P = 1$ and $Q = (1 - D)/4$, where D is the first integer in the sequence $\{5, -7, 9, -11, 13, -15, \dots\}$ for which $\text{GCD}(D, N) = 1$ and the Jacobi symbol $(D/N) = -1$. Note, however, that if N is a perfect square, no such D will exist, and the search for D would continue all the way to $\pm\sqrt{N}$, at which point $\text{GCD}(D, N) = \sqrt{N}$ would expose N as composite. Consequently, the algorithm also presumes the presence of a preliminary check for perfect squares (as well as even integers and integers < 3).

Methodology

Data Structures Used:

1. Arrays
2. Loops
3. Iteration
4. Recursion

Algorithm Techniques Employed:

1. Brute Force:
This was used to check each and every possible ways to check if the number is a pseudoprime or not.
2. Divide and Conquer:
Here, every problem in case of both Lucas Pseudoprime and Fermat was divided into smaller modules and the output was displayed based on the conditions satisfied.
3. Greedy:
Here, we greedily chose the numbers to be checked and tested for pseudo-primality.

Coding Language used:

The coding language used was Python. This language is easy to code and it is capable of displaying very long numbers.

Limitation to the algorithms:

1. Sometimes, iteration for a huge number was not possible.
2. Also, recursion also had a certain limit up to which it cannot recurse and hence the data got lost and program showed some errors.

Procedure Followed:

1. Perform a base-2 pseudoprime test on given number n , if the test fails, then n can be declared as a composite number and we can halt the procedure. But if the procedure is a success, then there is a possibility that the number is a prime number. So after this we proceed to step 2.
2. In step 2, we first find the number D for while the Jacobi symbol is -1 . Then after that we perform Lucas's pseudoprime test with discriminant D on n . If this test fails, then the number is declared as a composite number and we can halt the procedure. But if the procedure is a success, then there is a very high possibility that the number n is prime.

Algorithms

For Lucas Pseudoprime:

START

1: Set d, p, q as 5,1 and -1 respectively and $a_1=1.618$ and $b_e=-0.2472$

2: Define a function isSquare(x)

 If $x==1$, return 1

 Set low as 0 and high as integer form of $x/2$

 While $root*root!=x$

 Set $root=\text{integer form of } (low+high)/2$

 If $low+1 \geq high$, return 0

 Else if $root*root > x$, set $high=root$

 Else, set $low=root$

 Return 1

3: Define function js(a,p) which is used to calculate and find the Jacobi symbol.

 If $(a \% p == 0)$, return 0

 Else if $(\text{isSquare}(a \% p) == 1)$, return 1

 Else return -1

4: Define function U(n) which returns the value of U_n , where n is the coefficient using

Recursion

 If $n==0$, return 0,

 Else if $n==1$, return 1

 Else return $p*U(n-1)*U(n-2)$

5: Define function u(k). This function is similar to U(n), the only difference is this process is iterative

 Return $(a_1^k - (b_e^k)/(a_1 - b_e))$

6: Define function isPrime(n)

 If $n==2$, return true

 Else if $n==3$, return true

 Else if $n \% 2 == 0$, return False

 Else if $n \% 3 == 0$, return False

 Set i,w as 5,2

 While $i*i \leq n$

 If $n \% i == 0$, return False

$i+=w$

$w=6-w$

 return True

7: In order to calculate the time, store the present time in t

8: Set num and nos as 3 and 0 respectively

9: while $nos < 10$

 Set delta as $n - js(d,n)$ and $a = u(delta) \% n$

 If $a == 0$

 If $\text{prime}(n) == 0$ and $\text{gcd}(n,q) == 1$

 Print n and set $nos = nos + 1$

```
        Set num=num+1
10. Set b as the current time
11: Display t-b
STOP
```

For Strong Fermat Test:

START

1: Define function isprime(n) to check if the number is prime or not

```
    If n equals 2, return True
    Else if n equals 3, return True
    Else if n % 2 equals 0, return False
    If n % 3 equals 0, return False
    i = 5, w = 2
    while i * i <= n:
        if n % i equals 0, return False
        i += w
        w = 6 - w
    return True
```

2: Define function find_d(n) to find the value of d and r

(Note that: n is odd so n-1 is even. we have to find d such that $n-1 = d \cdot 2^r$)

```
    count=0, m=n-1
    while (m%2 equals 0)
        m=integer form of m/2
        count=count+1
    d=m, s=count
    return d,s
```

3: num=3, a=2

4: set t as the current time to calculate the time taken to perform the program

5: While nos < 10

```
    if(not isprime(num))
        d,s=find_d(num)
        for r in range(0,s)
            p=d*(2^r)
            if ((pow(a,d,num)==0)), print(num)
            else if((pow(a,p,num)==num-1)), print(num)
            nos=nos+1
        num=num+2
```

6: b=time.time()which stores the current time in b

7: Display (b-t)

STOP

Codes

Lucas Pseudoprimes

Code 1:

```
#Fastest and most efficient
from random import randint
import time
from fractions import gcd
from math import sqrt
def isprime(n): # The function returns true if n is prime else false
    #Returns True if n is prime.
    if n == 2:
        return True
    if n == 3:
        return True
    if n % 2 == 0:
        return False
    if n % 3 == 0:
        return False
    i = 5
    w = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += w
        w = 6 - w
    return True

#Code to tabulate first nos pseudoprimes where nos is given by the user
d=5 # d is the discriminant i.e.  $p^2-4q$ 
p=1
q=-1
al= (p+(d**0.5))*0.5 # Roots of the equation
be= (p-(d**0.5))*0.5 #  $x^2-px+q$ 
def js(a,p): # To find the jacobian symbol
    if(a%p==0):
        return 0
    elif (isSquare(a%p)==1):
        return 1
    else:
        return -1
def u(n): #To find Un using iteration
    v1, v2, v3 = 1, 1, 0 #Initialise a matrix [[1,1],[1,0]]
```



```
    for rec in bin(n)[3:]: #Perform fast exponentiation of the matrix (quickly raise it to the nth power)
```

```
        calc = v2*v2
```

```
        v1, v2, v3 = v1*v1+calc, (v1+v3)*v2, calc+v3*v3
```

```
        if rec=='1': v1, v2, v3 = v1+v2, v1, v2
```

```
    return v2
```

```
def U(n): #To find Un using recursion
```

```
    if(n==0):
```

```
        return 0
```

```
    elif(n==1):
```

```
        return 1
```

```
    else:
```

```
        return p*U(n-1)-q*U(n-2)
```

```
def isSquare(x): #To check whether a number is a perfect square
```

```
    if x == 1:
```

```
        return 1
```

```
    low = 0
```

```
    high = x // 2
```

```
    root = high
```

```
    while root * root != x:
```

```
        root = (low + high) // 2
```

```
        if low + 1 >= high:
```

```
            return 0
```

```
        if root * root > x:
```

```
            high = root
```

```
        else:
```

```
            low = root
```

```
    return 1
```

```
def lucas(n): #Returns true if the number is Lucas else returns False
```

```
    if(not isprime(n)):
```

```
        if(gcd(n,5)==1):
```

```
            delta=n+1
```

```
            a=u(delta)%n
```

```
            if(a==0):
```

```
                return True
```

```
    return False
```

```
nos=0
```

```
no=3
```

```
t=time.time()
```

```
while(nos<10): #main function to tabulate lucas pseudoprime numbers
```

```
    #if(fermat_1(num) or fermat_2(num)):
```

```
    if(lucas(no)):
```

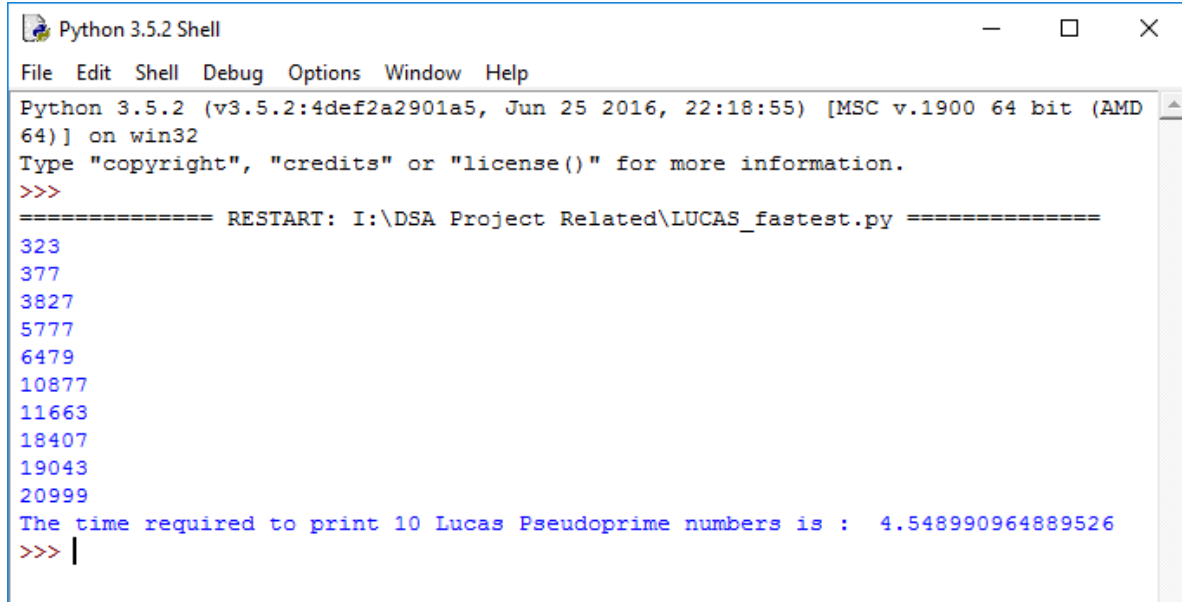
```
        print(no)
```

```
        nos=nos+1
```

```
    no=no+2
```

```
b=time.time()
print("The time required to print",nos, "Lucas Pseudoprime numbers is : ",(b-t))
```

Screenshot:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: I:\DSA Project Related\LUCAS_fastest.py =====
323
377
3827
5777
6479
10877
11663
18407
19043
20999
The time required to print 10 Lucas Pseudoprime numbers is : 4.548990964889526
>>> |
```

Code 2:

#Fastest and moderately efficient

```
from fractions import gcd
from math import sqrt
import time
```

```
d=5
p=1
q=-1
a1= (p+(d**0.5))*0.5
be= (p-(d**0.5))*0.5
```

```
def isSquare(x):
    if x == 1:
        return 1
    low = 0
    high = x // 2
    root = high
    while root * root != x:
        root = (low + high) // 2
        if low + 1 >= high:
            return 0
        if root * root > x:
```

```
    high = root
else:
    low = root
return 1
```

```
def js(a,p):
    if(a%p==0):
        return 0
    elif (isSquare(a%p)==1):
        return 1
    else:
        return -1
```

```
def U(n):
    if(n==0):
        return 0
    elif(n==1):
        return 1
    else:
        return p*U(n-1)-q*U(n-2)
```

```
def u(n):
    v1, v2, v3 = 1, 1, 0  #Initialise a matrix [[1,1],[1,0]]
    for rec in bin(n)[3:]: #Perform fast exponentiation of the matrix (quickly raise it to the nth
power)
        calc = v2*v2
        v1, v2, v3 = v1*v1+calc, (v1+v3)*v2, calc+v3*v3
        if rec=='1':    v1, v2, v3 = v1+v2, v1, v2
    return v2
```

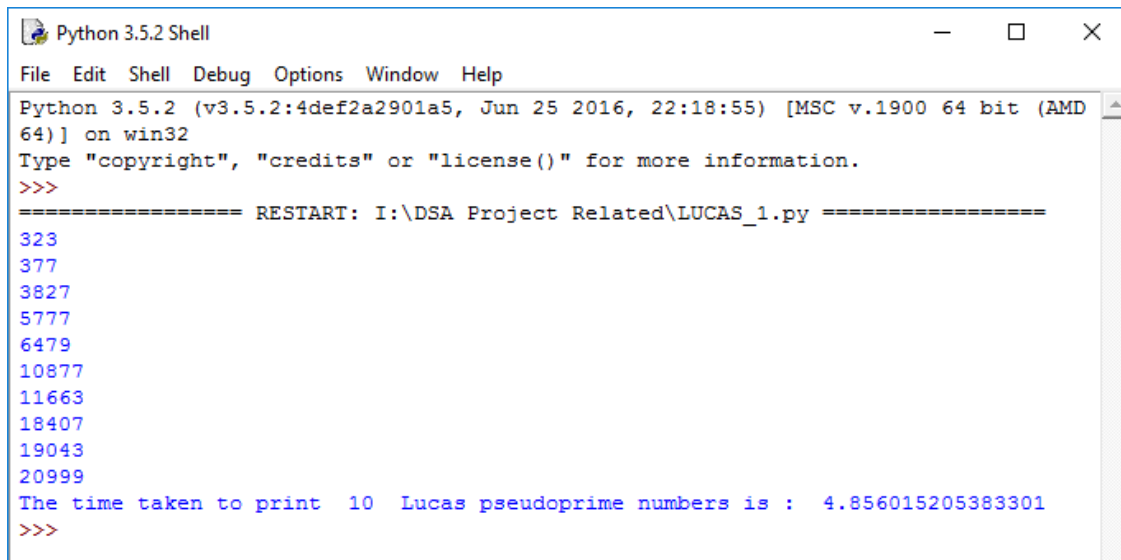
```
def prime(num):
    if n==2:
        return True
    if n==3:
        return True
    if n%2==0:
        return False
    if n%3==0:
        return False
    i=5
    w=2
    while(i*i<=n):
        if(n%i==0):
            return False
        i+=w
        w=6-w
```

```

    return True
nos=0
n=3
t=time.time()
#for n in range(2,100000):
while(nos<10):
    if(not prime(n)):
        t1=(time.time())*60
        if(gcd(n,5)==1):
            delta=n+1
            a=u(delta)%n
            if(a==0):
                print(n)
                nos=nos+1
    n=n+2
t2=time.time()
T=t2-t
print("The time taken to print ",nos," Lucas pseudoprime numbers is : ",T)

```

Screenshot:



```

Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD
64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: I:\DSA Project Related\LUCAS_1.py =====
323
377
3827
5777
6479
10877
11663
18407
19043
20999
The time taken to print 10 Lucas pseudoprime numbers is : 4.856015205383301
>>>

```

Code 3:

#Slowest and most straightforward method

```

from fractions import gcd
from math import sqrt
import time

```

```

d=5
p=1
q=-1

```

```
al= (p+(d**0.5))*0.5
be= (p-(d**0.5))*0.5
```

```
def isSquare(x):
    if x == 1:
        return 1
    low = 0
    high = x // 2
    root = high
    while root * root != x:
        root = (low + high) // 2
        if low + 1 >= high:
            return 0
        if root * root > x:
            high = root
        else:
            low = root
    return 1
```

```
def js(a,p):
    if(a%p==0):
        return 0
    elif (isSquare(a%p)==1):
        return 1
    else:
        return -1
```

```
def U(n):
    if(n==0):
        return 0
    elif(n==1):
        return 1
    else:
        return p*U(n-1)-q*U(n-2)
```

```
def u(n):
    v1, v2, v3 = 1, 1, 0 # initialise a matrix [[1,1],[1,0]]
    for rec in bin(n)[3:]: # perform fast exponentiation of the matrix (quickly raise it to the nth
power)
        calc = v2*v2
        v1, v2, v3 = v1*v1+calc, (v1+v3)*v2, calc+v3*v3
        if rec=='1': v1, v2, v3 = v1+v2, v1, v2
    return v2
```

```
def prime(num):
    if n==2:
```

```

        return True
    if n==3:
        return True
    if n%2==0:
        return False
    if n%3==0:
        return False
    i=5
    w=2
    while(i*i<=n):
        if(n%i==0):
            return False
        i+=w
        w=6-w
    return True
nos=0
n=3
t=time.time()
#for n in range(2,100000):
while(nos<10):
    if(not prime(n)):
        t1=(time.time())*60
        if(gcd(n,5)==1):
            delta=n+1
            a=u(delta)%n
            if(a==0):
                print(n)
            nos=nos+1
    n=n+2
t2=time.time()
T=t2-t
print("The time taken to print ",nos," Lucas pseudoprime numbers is : ",T, "seconds")

```

Screenshot:

```
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD
64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: I:\DSA Project Related\LUCAS_2_slow.py =====
323
377
3827
5777
6479
10877
11663
18407
19043
20999
The time taken to print 10 Lucas pseudoprime numbers is : 5.126712322235107 se
conds
>>>
```

Base 2 Fermat Test:

Code 1:

#Code to input a no and check for fermat primality test

```
from random import randint
```

```
import time
```

```
"""def modpow(x,n,m):
```

```
    if n==0:
```

```
        return 1
```

```
    elif n==1:
```

```
        return x
```

```
    elif n%2==0:
```

```
        return modpow(x*(x%m),n//2,m)%m
```

```
    elif n%2==1:
```

```
        return (x*modpow(x*(x%m),(n-1)// 2,m)%m)%m"""
```

```
def isprime(n):
```

```
    """Returns True if n is prime."""
```

```
    if n == 2:
```

```
        return True
```

```
    if n == 3:
```

```
        return True
```

```
if n % 2 == 0:
    return False
if n % 3 == 0:
    return False
```

```
i = 5
w = 2
```

```
while i * i <= n:
    if n % i == 0:
        return False
```

```
i += w
w = 6 - w
```

```
return True
num=2047
```

```
def find_d(n):
    """if(n%2==0):
        print("Number must be odd")
        return 0,0"""
```

#this function returns value of d and r

#n is odd so n-1 is even. we have to find d such that $n-1=d \cdot 2^r$

```
count=0;
m=n-1;
while(m%2==0):
    m=m//2;
    count=count+1;
d=m;
r=count;
return d,r
```

#Code to tabulate first 10 pseudoprimes

```
nos=1;
flag=True
t=time.time()
while(nos<=10):
    a=2#we have specified a as 2 because we are checking only for strong base 2
    pseudoprime
    if(not isprime(num)):
        d1,r1=find_d(num)
```

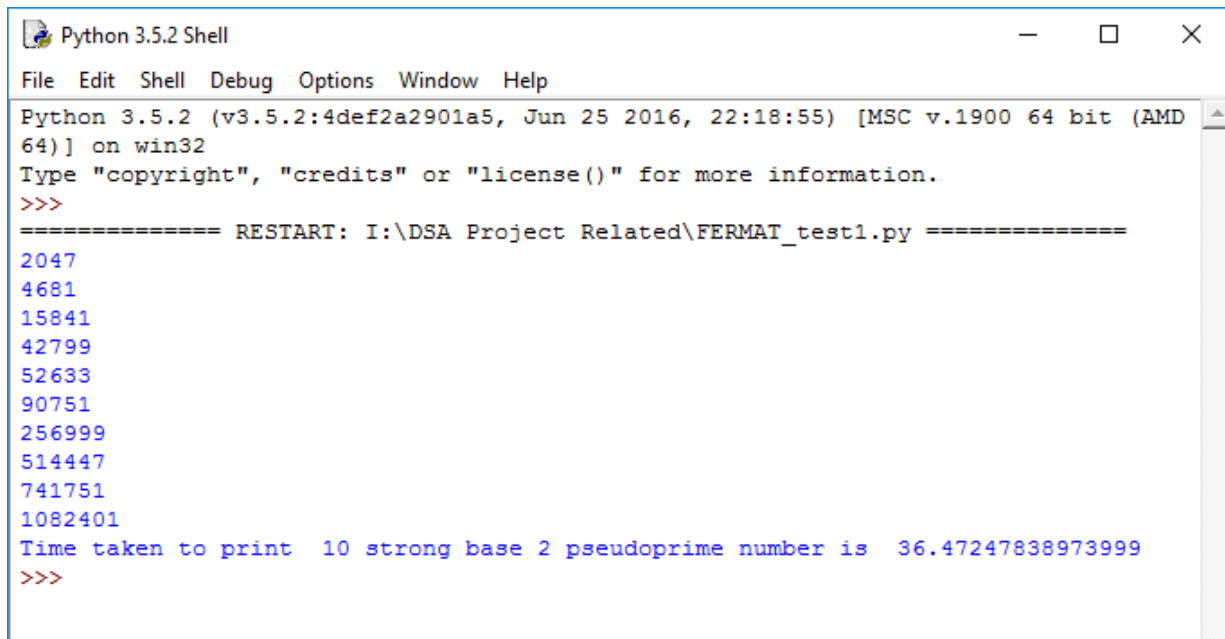


```

        if (pow(a,d1,num)==1):
            print(num)
            nos=nos+1
            num=num+2
        num=num+2
b=time.time()
print("Time taken to print ", nos-1, "strong base 2 pseudoprime number is ", (b-t))
STOP

```

Screenshot:



```

Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD
64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: I:\DSA Project Related\FERMAT_test1.py =====
2047
4681
15841
42799
52633
90751
256999
514447
741751
1082401
Time taken to print 10 strong base 2 pseudoprime number is 36.47247838973999
>>>

```

Code 2:

#Code to Check for fermat primality test

```

from random import randint
import time
"""def modpow(x,n,m):
    if n==0:
        return 1
    elif n==1:
        return x
    elif n%2==0:
        return modpow(x*(x%m),n//2,m)%m
    elif n%2==1:
        return (x*modpow(x*(x%m),(n-1)// 2,m)%m)%m"""

```

```

def isprime(n):
    # the function returns true if n is prime else it returns false
    """Returns True if n is prime."""
    if n == 2:
        return True
    if n == 3:
        return True
    if n % 2 == 0:
        return False
    if n % 3 == 0:
        return False

    i = 5
    w = 2

    while i * i <= n:
        if n % i == 0:
            return False

        i += w
        w = 6 - w

    return True

def find_d(n):
    #this function returns value of d and r
    #n is odd so n-1 is even. we have to find d such that n-1=d.2^r
    count=0; #count stores the power of r
    m=n-1;
    while(m%2==0):
        m=m//2;
        count=count+1;
    d=m;
    s=count;
    return d,s
#Code to tabulate first nos pseudoprimes, nos is given by the user
num=3 # the starting number is set as 3
a=2
nos=0 # a counter to count number of prime numbers
t=time.time()

```

```

#we have specified a as 2 because we are checking only for strong base 2
pseudoprime
while(nos<20):
    if(not isprime(num)): # we are eliminating prime numbers as pseudoprimes are
        composite nos
        d,s=find_d(num)
        for r in range(0,s):
            p=d*(2**r)
            if((pow(a,p,num)==num-1)):
                print(num)
                nos=nos+1
                num=num+2
b=time.time() #stores the current time
t2=b-t # stores the time to find nos numbers
print("The time taken to print ",nos," Strong base 2 pseudoprime numbers is : ",t2,
"seconds")

```

Screenshot:

```

Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD
64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: I:\DSA Project Related\FERMAT_test2.py =====
3277
4033
8321
29341
49141
65281
74665
80581
85489
88357
104653
130561
196093
233017
252601
253241
280601
314821
357761
390937
The time taken to print 20 Strong base 2 pseudoprime numbers is : 20.714864969
25354 seconds
>>>

```

Baillie- PSW Test:

```
from random import randint
import time
from fractions import gcd
from math import sqrt
"""def modpow(x,n,m):
    if n==0:
        return 1
    elif n==1:
        return x
    elif n%2==0:
        return modpow(x*(x%m),n//2,m)%m
    elif n%2==1:
        return (x*modpow(x*(x%m),(n-1)// 2,m)%m)%m"""
def isprime(n): # The function returns true if n is prime else false
    """Returns True if n is prime."""
    if n == 2:
        return True
    if n == 3:
        return True
    if n % 2 == 0:
        return False
    if n % 3 == 0:
        return False
    i = 5
    w = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += w
        w = 6 - w
    return True

num=2047
def find_d(n):
    """if(n%2==0):
        print("Number must be odd")
        return 0,0"""
```

```

#this function returns value of d and r
#n is odd so n-1 is even. we have to find d such that  $n-1=d \cdot 2^r$ 
count=0;
m=n-1;
while(m%2==0):
    m=m//2;
    count=count+1; #count stores d power of 2
d=m;
r=count;
return d,r

#Code to tabulate first nos pseudoprimes where nos is given by the user
def fermat_1(num):
    a=2#we have specified a as 2 because we are checking only for strong base 2
    pseudoprime
    if(not isprime(num)):
        d1,r1=find_d(num)
        if (pow(a,d1,num)==1):#if  $(a^d) \% num$  equals 0 then it is a fermat
        pseudoprime number
        return True
    return False
def fermat_2(num):
    a=2
    if(not isprime(num)): # we are eliminating prime numbers as pseudoprimes are
    composite nos
        d,s=find_d(num)
        for r in range(0,s):
            p=d*(2**r)
            if((pow(a,p,num)==num-1)):
                return True
        return False
d=5 # d is the discriminant i.e.  $p^2-4q$ 
p=1
q=-1
al= (p+(d**0.5))*0.5 # roots of the equation
be= (p-(d**0.5))*0.5 # $x^2-px+q$ 
def js(a,p): # to find the jacobian symbol
    if(a%p==0):
        return 0
    elif (isSquare(a%p)==1):
        return 1

```

```

else:
    return -1
def u(n): #to find Un using iteration
    v1, v2, v3 = 1, 1, 0 # initialise a matrix [[1,1],[1,0]]
    for rec in bin(n)[3:]: # perform fast exponentiation of the matrix (quickly raise it
to the nth power)
        calc = v2*v2
        v1, v2, v3 = v1*v1+calc, (v1+v3)*v2, calc+v3*v3
        if rec=='1': v1, v2, v3 = v1+v2, v1, v2
    return v2
def U(n): # to find Un using recursion
    if(n==0):
        return 0
    elif(n==1):
        return 1
    else:
        return p*U(n-1)-q*U(n-2)

```

```

def isSquare(x): # to check whether a number is a perfect square
    if x == 1:
        return 1
    low = 0
    high = x // 2
    root = high
    while root * root != x:
        root = (low + high) // 2
        if low + 1 >= high:
            return 0
        if root * root > x:
            high = root
        else:
            low = root
    return 1
def lucas(n): # returns true if the number is Lucas else returns False
    if(not isprime(n)):
        if(gcd(n,5)==1):
            delta=n+1
            a=u(delta)%n
            if(a==0):

```

```
        return True
    return False
nos=0
no=3
t=time.time()
while(nos<10): #main function to tabulate lucas pseudoprime numbers
    #if(fermat_1(num) or fermat_2(num)):
    if(fermat_1(num) or fermat_2(num)):
        if(lucas(num)):
            print(num)
            nos=nos+1
    no=no+2
b=time.time()
print("The time required to print ",nos, "Fermat Pseudoprime numbers is : ",(b-t))
```