Gregory Knapp

CS 312

Dr. Grimsman

9/14/2021

Project 1 Fermat Report

Time and Space Complexity Analysis of Project 1

The time complexity for each function found in fermat.py of the Project 1 code is as follows and is referenced from the comments preceding each function in fermat.py.

mod_exp(x, y, N) Complexity – Lines 12-19

```
# One function call of mod_exp is Time Complexity O(n^2)
# Because we iterate based on y // 2, we get approximately log y recursive
calls of mod_exp
# Assuming y has a similar number of bits as x, we can approximate that log y
= O(n)
#
# Therefore, mod_exp total Time Complexity = O(n^3) or in other words O(n^2)
for the function, running O(n) times.
#
# mod_exp Space Complexity = O(n) as it has n levels of recursion, so the
computer return n times
# and the stack will grow with each recursive call.
```

fprobability(k) – Lines 38 – 44

```
# fprobability(k) uses k to make one calculation for the probability and
returns it. The function
# uses the ** power operator to calculate 2^k which runs in O(n^2) time.
#
# Because the rest of equation runs in O(1) time, the time complexity for
calculating
# probability and the whole function is O(n^2)
#
# The space complexity is O(1) as all the program is doing is creating one
variable and returning it.
```

mprobability(k) – Lines 52 – 58

```
# mprobability(k) uses k to make one calculation for the probability and
returns it. The function
# is simple, but includes the ** power operator which runs in O(n^2) time.
```

```
#
# Because the rest of equation runs in O(1) time, the time complexity for
calculating
# probability and the whole function is O(n^2)
#
# The space complexity is O(1) as all the program is doing is creating one
variable and returning it.
```

fermat(N, k) – Lines 66 – 73

```
# fermat(N, k) uses mod_exp of Time Complexity O(n^3) on each value of an
array of length k, making that portion of
# the algorithm run in O(k*n^3) times, where k is the number of loops the for
loop is making to test each of the
# random values. There is another for loop before used to generate the values
in O(k) times where k is the number
# of values to generate. This is not significant enough to change the result
of the Time Complexity analysis.
#
# Therefore, Time Complexity of fermat is O(n^3)
#
# Space Complexity of fermat is O(n) as it only creates the one-dimensional
array to store test values.
```

miller_rabin(N,k) – Lines 100 – 109

```
# In miller_rabin, most of the Time Complexity stems from calling mod_exp
during the while loop to check the test
# values. mod_exp has a Time Complexity of O(n^3). There are three loops
within the function, a for loop to set the
# random test values, a for loop to test each value in the array, and a while
loop to take the square root of the
# exponential function after running mod_exp to test for primality. Both for
loops run in O(k) time where k is the
# number of values to test and the while loop runs in O(log n) time, as it is
being divided by 2 with each iteration
# of the loop. However, all of the time complexity of the loop is dominated
by the O(n^3) mod_exp function call.
#
# Therefore, the resulting time complexity of the miller_rabin test function
is O(n^3).
#
# Space Complexity is O(n) as the function only creates the one dimensional
array test_values to store our values
```

Algorithm Disagreement

Although the two algorithms have the same purpose of testing for probable primality, there is a key difference in their ability to accurately determine what is a prime number. Fermat's little theorem is not given as an if and only if, but only if, because it is stating the behavior of a prime number, not a composite one. Due to this, Carmichael Numbers (561, 1105, etc.) can mimic the behavior of a prime number according to Fermat's Little Theorem and pass off as a prime number when they are composite. The Miller-Rabin test builds off Fermat's theorem with a safety against Carmichael numbers by checking for the first instance of variation from the result of 1 mod N.

Knowing this from class, I went in and tested first numbers that I knew were prime (73, 101) as well as numbers that were obviously composite (even numbers, 55) to make sure that the base functionality of the two tests were working. From there, I added in one of the known Carmichael numbers for N, 561, to see if the Fermat test would accidentally pass it. If I ran the algorithm with k = 10, the Fermat's test would almost always determine that 561 was composite, but with a lower number of tests, it was quite often that the randomly selected test values would result in Fermat's test passing 561. On the other hand, the Miller-Rabin test would return not prime for 561, resulting in the discrepancy as seen in Screenshot #1 and #2 below.

Discussion on Probability Equations

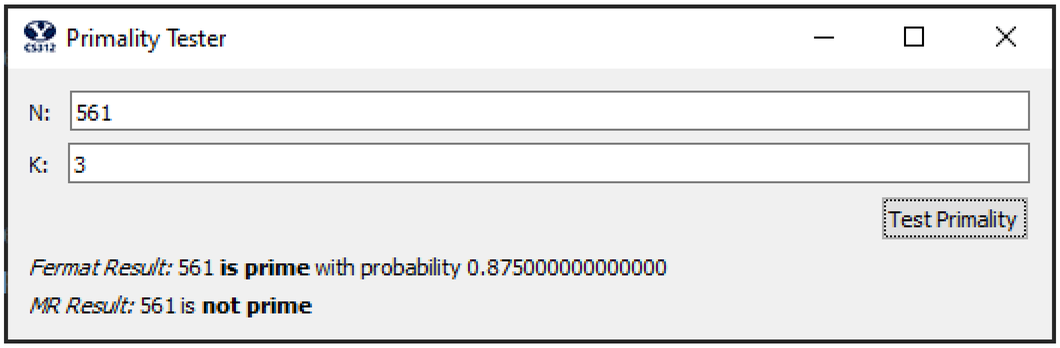
For each test, there is not a 100% certainty in the result due to the random selection of a values for testing. As a result, we can say that we are calculating the probability of the test determining that is prime. As a result, there is a measure of a one-sided error that the probability equations considers.

For Fermat's little theorem, the textbook states that Fermat's theorem has <= ½ chance of returning a false positive result. Since we are running the test k times, we can then take the probability of the one-sided error as 1 / 2^k. Since this is the probability of having a false positive test, the probability function will then subtract the result from 1, resulting in the probability that is displayed at the end of the test.

The Miller-Rabin test behaviors very similarly, only it has a slightly different one-sided error probability. The textbook states on page 28 that "at least three-forth of the possible values of a between 1 and N-1 will reveal a composite N, even if it's a Carmichael Number." (Dasgupta, 2008). Since ¾ of the results will result in a composite result, the remaining probability of ¼ can be considered as the possibility of a one-sided error, which is reduced with the number of tests run. Our probability calculation then becomes 1 – (1/4^k), resulting in the final probability displayed on the screen.

Screenshots

Screenshot #1 – Disagreement between the two tests when N = 561 and K = 3



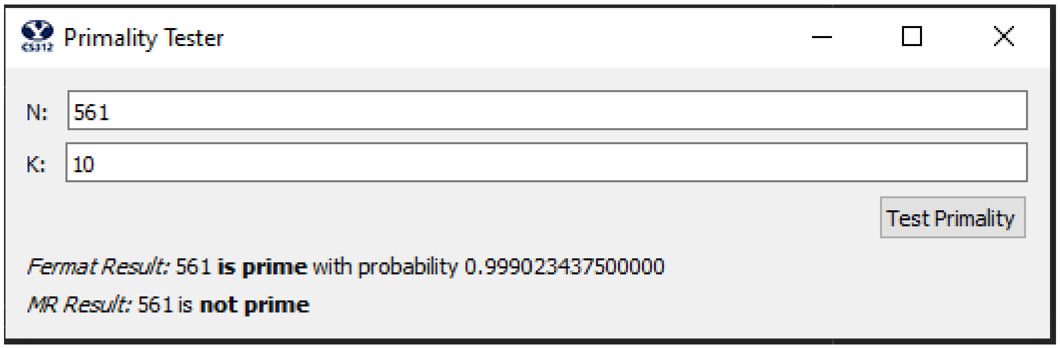
N: 561

K: 3

Fermat Result: 561 **is prime** with probability 0.875000000000000

MR Result: 561 is **not prime**

Screenshot #2 Disagreement between the two tests when N = 561 and K = 10


Primality Tester

N: 561

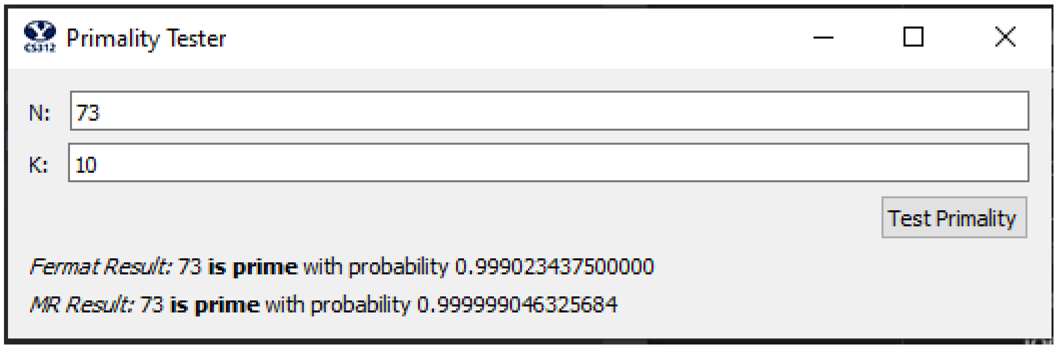
K: 10

Test Primality

Fermat Result: 561 **is prime** with probability 0.999023437500000

MR Result: 561 is **not prime**

Screenshot #3 – A working example of a prime number using N = 73


Primality Tester

N: 73

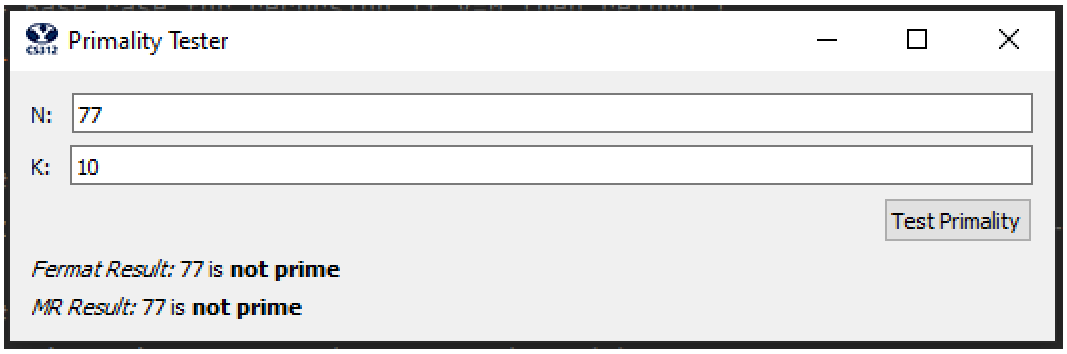
K: 10

Test Primality

Fermat Result: 73 **is prime** with probability 0.999023437500000

MR Result: 73 **is prime** with probability 0.999999046325684

Screenshot #4 – A working example of a composite number using N = 100


Primality Tester

N: 77

K: 10

Test Primality

Fermat Result: 77 is **not prime**

MR Result: 77 is **not prime**

Source Code Files

fermat.py code file

```
import random


# While for most cases the two algorithms should return the same result for
prime_test, there is one major difference
# between the two that will cause some disagreements. As is mentioned in the
Algorithms textbook, Fermat's theorem is
# an if statement, not if and only if and
def prime_test(N, k):
    # This is main function, that is connected to the Test button. You don't
need to touch it.
    return fermat(N, k), miller_rabin(N, k)


# One function call of mod_exp is Time Complexity O(n^2)
# Because we iterate based on y // 2, we get approximately log y recursive
calls of mod_exp
# Assuming y has a similar number of bits as x, we can approximate that log y
= O(n)
#
# Therefore, mod_exp total Time Complexity = O(n^3) or in other words O(n^2)
for the function, running O(n) times.
#
# mod_exp Space Complexity = O(n) as it has n levels of recursion, so the
computer return n times
# and the stack will grow with each recursive call.
def mod_exp(x, y, N):

    # Base case for recursion if y=0 then return 1
    if y == 0:  # Time Complexity O(1)
        return 1  # Time Complexity O(1)

    # Recursive mod_exp call x = x, y = y//2, N = N
    z = mod_exp(x, (y // 2), N)  # Time Complexity for 1 mod_exp call -
O(n^2)

    # if y is even return z^2mod N
    if (y % 2) == 0:  # Time Complexity O(1)
        return (z ** 2) % N  # Time Complexity O(n^2) + O(n^2) + O(1) =
O(n^2)

    # if y is odd return x*z^2 mod N
    else:  # Time Complexity O(1)
        return (x * (z ** 2)) % N  # Time Complexity O(n^2) + O(n^2) + O(n^2)
+ O(1) = O(n^2)


# fprobability(k) uses k to make one calculation for the probability and
returns it. The function
# uses the ** power operator to calculate 2^k which runs in O(n^2) time.
#
```

```
# Because the rest of equation runs in O(1) time, the time complexity for
calculating
# probability and the whole function is O(n^2)
#
# The space complexity is O(1) as all the program is doing is creating one
variable and returning it.
def fprobability(k):
    # Get probability of Fermat's Little Theorem using 1/(2^k)
    probability = 1 - (1 / (2 ** k))  # Time Complexity O(n^2)

    return probability  # Time Complexity O(1)


# mprobability(k) uses k to make one calculation for the probability and
returns it. The function
# is simple, but includes the ** power operator which runs in O(n^2) time.
#
# Because the rest of equation runs in O(1) time, the time complexity for
calculating
# probability and the whole function is O(n^2)
#
# The space complexity is O(1) as all the program is doing is creating one
variable and returning it.
def mprobability(k):
    # Get probability of M-R test results using 1/(4^k)
    probability = 1 - (1 / (4 ** k))  # Time Complexity O(n^2)

    return probability  # Time Complexity O(1)


# fermat(N, k) uses mod_exp of Time Complexity O(n^3) on each value of an
array of length k, making that portion of
# the algorithm run in O(k*n^3) times, where k is the number of loops the for
loop is making to test each of the
# random values. There is another for loop before used to generate the values
in O(k) times where k is the number
# of values to generate. This is not significant enough to change the result
of the Time Complexity analysis.
#
# Therefore, Time Complexity of fermat is O(n^3)
#
# Space Complexity of fermat is O(n) as it only creates the one-dimensional
array to store test values.
def fermat(N, k):
    # Check for even N and return composite immediately
    if N % 2 == 0:  # Time Complexity O(1)
        return 'composite'  # Time Complexity O(1)

    # Create empty array to store test values
    test_values = []  # Time Complexity O(1)

    # Generate random values (must be a < N) k times
    for i in range(k):  # Time Complexity O(n)
        test_values.append(random.randint(1, N - 1))  # Time Complexity O(1),
Space Complexity O(n)

    #  Loop through test values array - Time Complexity O(n)
    for value in test_values:

        # Run mod_exp for a=x, N-1 = y, and N = N
        fermat_result = mod_exp(value, N-1, N)  # Time Complexity O(n^3)

        # If =! 1 mod N, return composite, end for loop, if == 1 mod N,
continue for loop
        if fermat_result != 1:  # Time Complexity O(1)
            return 'composite'  # Time Complexity O(1)

    # return prime if all values pass fermat test
    return 'prime'  # Time Complexity O(1)


# In miller_rabin, most of the Time Complexity stems from calling mod_exp
during the while loop to check the test
# values. mod_exp has a Time Complexity of O(n^3). There are three loops
within the function, a for loop to set the
# random test values, a for loop to test each value in the array, and a while
loop to take the square root of the
# exponential function after running mod_exp to test for primality. Both for
loops run in O(k) time where k is the
# number of values to test and the while loop runs in O(log n) time, as it is
being divided by 2 with each iteration
# of the loop. However, all of the time complexity of the loop is dominated
by the O(n^3) mod_exp function call.
#
# Therefore, the resulting time complexity of the miller_rabin test function
is O(n^3).
#
# Space Complexity is O(n) as the function only creates the one dimensional
array test_values to store our values
def miller_rabin(N, k):
    # Check for even N and return composite immediately
    if N % 2 == 0:  # Time Complexity O(1)
        return 'composite'  # Time Complexity O(1)

    # Create empty array to store test values
    test_values = []  # Time Complexity O(1)

    # Generate random values (must be a < N) k times
    for i in range(k):  # Time Complexity O(n)
        test_values.append(random.randint(1, N - 1))  # Time Complexity O(1),
Space Complexity O(n)

    # Loop through test values array
    for value in test_values:  # Time Complexity O(n)

        # Set variable to track exponent y and result of Miller-Rabin test
        miller_rabin_result = 1  # Time Complexity O(1)
        y = N - 1  # Time Complexity O(1) TODO

        # Repeat Miller-Rabin test until exit condition met
        while miller_rabin_result == 1 and (y % 2) == 0:  # Time Complexity
O(log n)
            miller_rabin_result = mod_exp(value, y, N)  # Time Complexity
O(n^3)
```

```
            # Value passes M-R test if result == -1
            if miller_rabin_result == N - 1:  # Time Complexity O(1)
                break  # Time Complexity O(1)

            # Number is composite if M-R test != 1 or -1
            if miller_rabin_result != 1:  # Time Complexity O(1)
                return 'composite'  # Time Complexity O(1)

            # Square root x^y by dividing y in half
            y = y // 2  # Time Complexity O(1) TODO

    # Return prime if all test values pass M-R test
    return 'prime'  # Time Complexity O(1)
```

Proj1GUI.py

```
#!/usr/bin/env python3

import signal
import sys

#
# This grabs the correct version of PYQT and depends on the existence of a
file called
# "which_pyqt.py" that says which version to use.  Versions before PYQT4 are
not supported.
#
from which_pyqt import PYQT_VER
if PYQT_VER == 'PYQT5':
   from PyQt5.QtWidgets import QApplication, QWidget
   from PyQt5.QtGui import QIcon
   from PyQt5.QtWidgets import QHBoxLayout, QVBoxLayout
   from PyQt5.QtWidgets import QLabel, QPushButton, QLineEdit
elif PYQT_VER == 'PYQT4':
   from PyQt4.QtGui import QApplication, QWidget
   from PyQt4.QtGui import QHBoxLayout, QVBoxLayout
   from PyQt4.QtGui import QIcon, QLabel, QPushButton, QLineEdit
else:
   raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))


# Import in the code that implements the actual primality testing
from fermat import *
#from fermat_complete import *


class Proj1GUI( QWidget ):

   def __init__( self ):
      super().__init__()
      self.initUI()

   def initUI( self ):
      self.setWindowTitle('Primality Tester')
      self.setWindowIcon( QIcon('icon312.png') )

        # Now setup for project 1
      vbox = QVBoxLayout()
      self.setLayout( vbox )

      self.input_n = QLineEdit('312')
      self.input_k = QLineEdit('10')
      self.test    = QPushButton('Test Primality')
      self.outputF  = QLabel('<i>N is the number to test, K is how many
random trials</i>')
      self.outputF.setMinimumSize(500,0)
      self.outputMR = QLabel('')
      self.outputMR.setMinimumSize(500,0)

      # N
      h = QHBoxLayout()
      h.addWidget( QLabel( 'N: ' ) )
      h.addWidget( self.input_n )
      vbox.addLayout(h)

        # K
      h = QHBoxLayout()
      h.addWidget( QLabel( 'K: ' ) )
      h.addWidget( self.input_k )
      vbox.addLayout(h)

        # Test
      h = QHBoxLayout()
      h.addStretch(1)
      h.addWidget( self.test )
      vbox.addLayout(h)

        # Output
      h = QHBoxLayout()
      h.addWidget( self.outputF )
      vbox.addLayout(h)
      h = QHBoxLayout()
      h.addWidget( self.outputMR )
      vbox.addLayout(h)

        # When the Test button is clicked, call testClicked()
      self.test.clicked.connect(self.testClicked)
        # Do the same if enter is pressed in either input field
      self.input_n.returnPressed.connect(self.testClicked)
      self.input_k.returnPressed.connect(self.testClicked)

      self.show()

#
# This is the method connected to the Test Button.  It calls the actual
primality test code
# (which you will implement) from the "fermat.py" file.
#
   def testClicked( self ):
      try:
         # Make sure inputs are valid integers
         n = int( self.input_n.text() )
         k = int( self.input_k.text() )

         # This is the call to the pass-through function that gets your
results, from
         # both the Fermat and Miller-Rabin tests you will implement
         fermat,mr = prime_test(n,k)

         # Output results from Fermat and compute the appropriate error
bound, if necessary
         if fermat == 'prime':
            prob = fprobability(k)
            self.outputF.setText( '<i>Fermat Result:</i> {:d} <b>is prime</b>
with probability {:5.15f}'.format(n,prob) )
         else: # Should be 'composite'
            self.outputF.setText('<i>Fermat Result:</i> {:d} is <b>not
prime</b>'.format(n))
```

```
         # Output results from Miller-Rabin and compute the appropriate error
bound, if necessary
         if mr == 'prime':
            prob = mprobability(k)
            self.outputMR.setText( '<i>MR Result:</i> {:d} <b>is prime</b>
with probability {:5.15f}'.format(n,prob) )
         else: # Should be 'composite'
            self.outputMR.setText('<i>MR Result:</i> {:d} is <b>not
prime</b>'.format(n))

        # If inputs not valid, display an error
      except Exception as e:
         self.outputF.setText('<i>ERROR:</i> inputs must be integers!')




if __name__ == '__main__':
    # This line allows CNTL-C in the terminal to kill the program
   signal.signal(signal.SIGINT, signal.SIG_DFL)

   app = QApplication(sys.argv)
   w = Proj1GUI()
   sys.exit(app.exec())
```

which_pyqt.py

```
PYQT_VER = 'PYQT5'
```