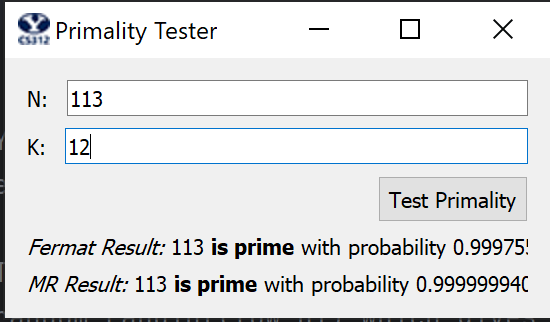
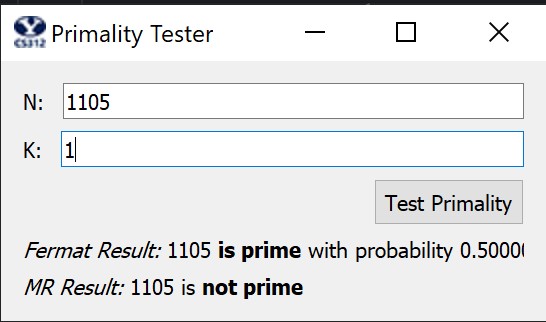
**Project 1 Report Alex Harmon**

1. See Appendix for code
2. Complexity
   1. ModExp
      1. Time
         1. n^3
         2. X \* z^2 % N requires two multiplies and a divide, the largest number of which is N, so N^2. Additionally, we must repeat this process N times. This leads to a total complexity of N^3
      2. Space
         1. The space complexity is N. You have to store a single value for each loop in the recursive call. Therefore, at most you have N units of storage
   2. Fermat
      1. Time
         1. K \* n^3 (you run through ModExp (O(n^3)) k times)
      2. Space
         1. The space complexity is N. You have to store a single value for each call to ModExp. Therefore, at most you have N units of storage
3. 
4. I chose 1105 as my input, because it’s a Carmichael number, meaning that it will pass the Fermat test because it shares the modular exponentiation behavior of prime numbers. However, because the Miller-Rabin test looks at a different aspect of prime numbers that Carmichael numbers DON’T share, it can weed out the outliers



1. For Fermat, the error rate is .5^k where k is the number of random numbers you try as the X value. For Miller-Rabin, the error rate is .25^k where k is the number of random numbers you try as the X value. The reason for this decrease in error is the elimination of Carmichael numbers giving false positives to the Fermat test

**Appendix**

import math

import random

# This is the main function that is connected to the Test button. You don't need to touch it.

def prime\_test(N, k):

return fermat(N,k), miller\_rabin(N,k)

# You will need to implement this function and change the return value.

def mod\_exp(x, y, N):

if y == 0:

return 1

z = mod\_exp(x, math.floor(y/2), N)

if (y % 2) == 0:

return pow(z, 2) % N

else:

return (x \* pow(z, 2)) % N

# You will need to implement this function and change the return value.

def fprobability(k):

return 1 - pow(.5, k)

# You will need to implement this function and change the return value.

def mprobability(k):

return 1 - pow(.25, k)

# You will need to implement this function and change the return value, which should be

# either 'prime' or 'composite'.

#

# To generate random values for a, you will most likely want to use

# random.randint(low,hi) which gives a random integer between low and

# hi, inclusive.

def fermat(N,k):

for i in range(k):

randomA = random.randint(2, N - 1)

if(mod\_exp(randomA,N-1, N) != 1):

return 'composite'

return 'prime'

# You will need to implement this function and change the return value, which should be

# either 'prime' or 'composite'.

#

# To generate random values for a, you will most likely want to use

# random.randint(low,hi) which gives a random integer between low and

# hi, inclusive.

def miller\_rabin(N,k):

for \_ in range(k):

a = random.randint(1, N - 1)

exponent = N - 1

ret = mod\_exp(a, exponent, N)

if ret == N - 1:

return 'prime'

elif ret != 1:

return 'composite'

while True:

if exponent % 2 == 0:

exponent /= 2

else:

break

ret = mod\_exp(a, exponent, N)

if ret == N - 1:

return 'prime'

elif ret != 1:

return 'composite'

return 'prime'