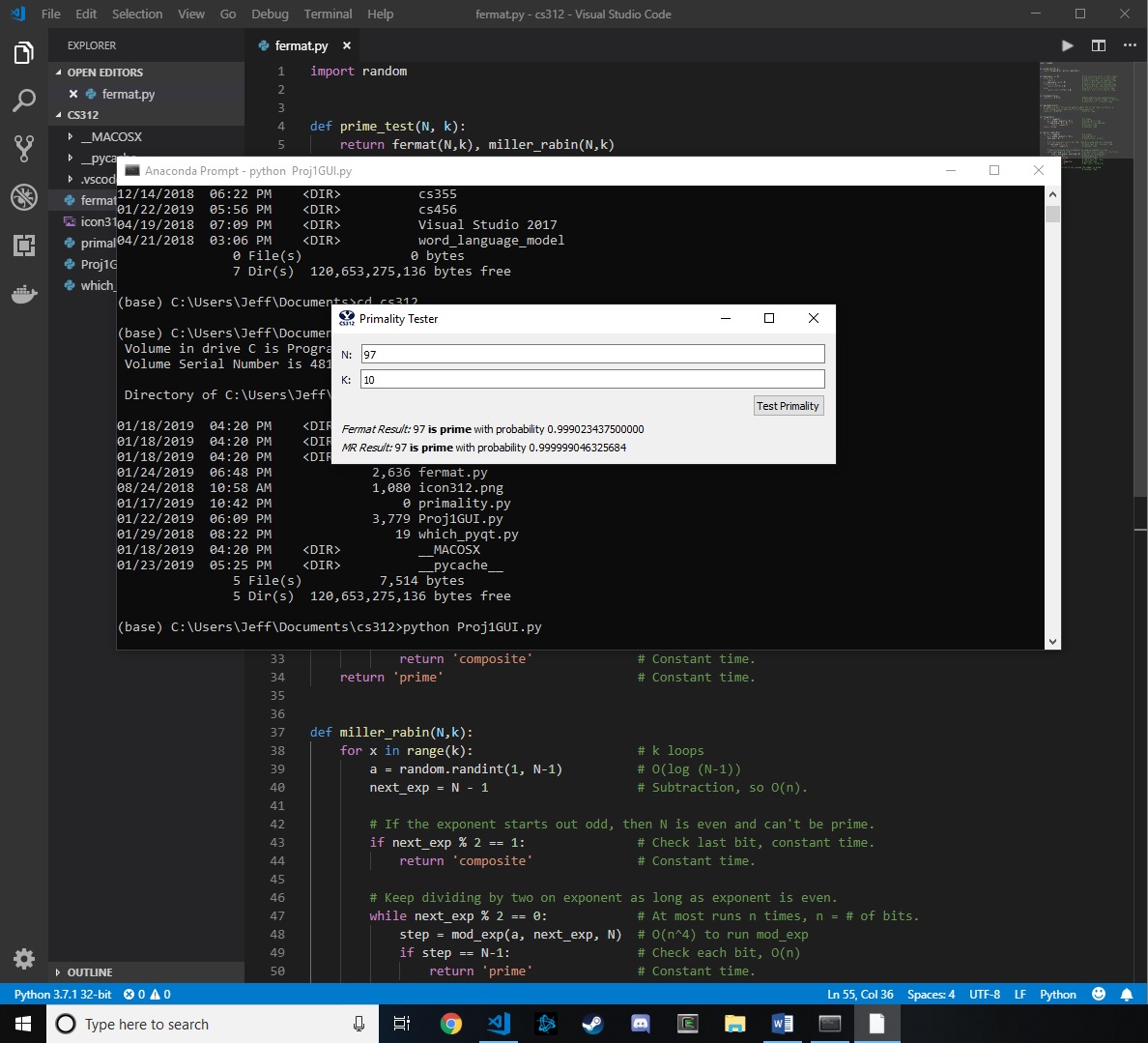
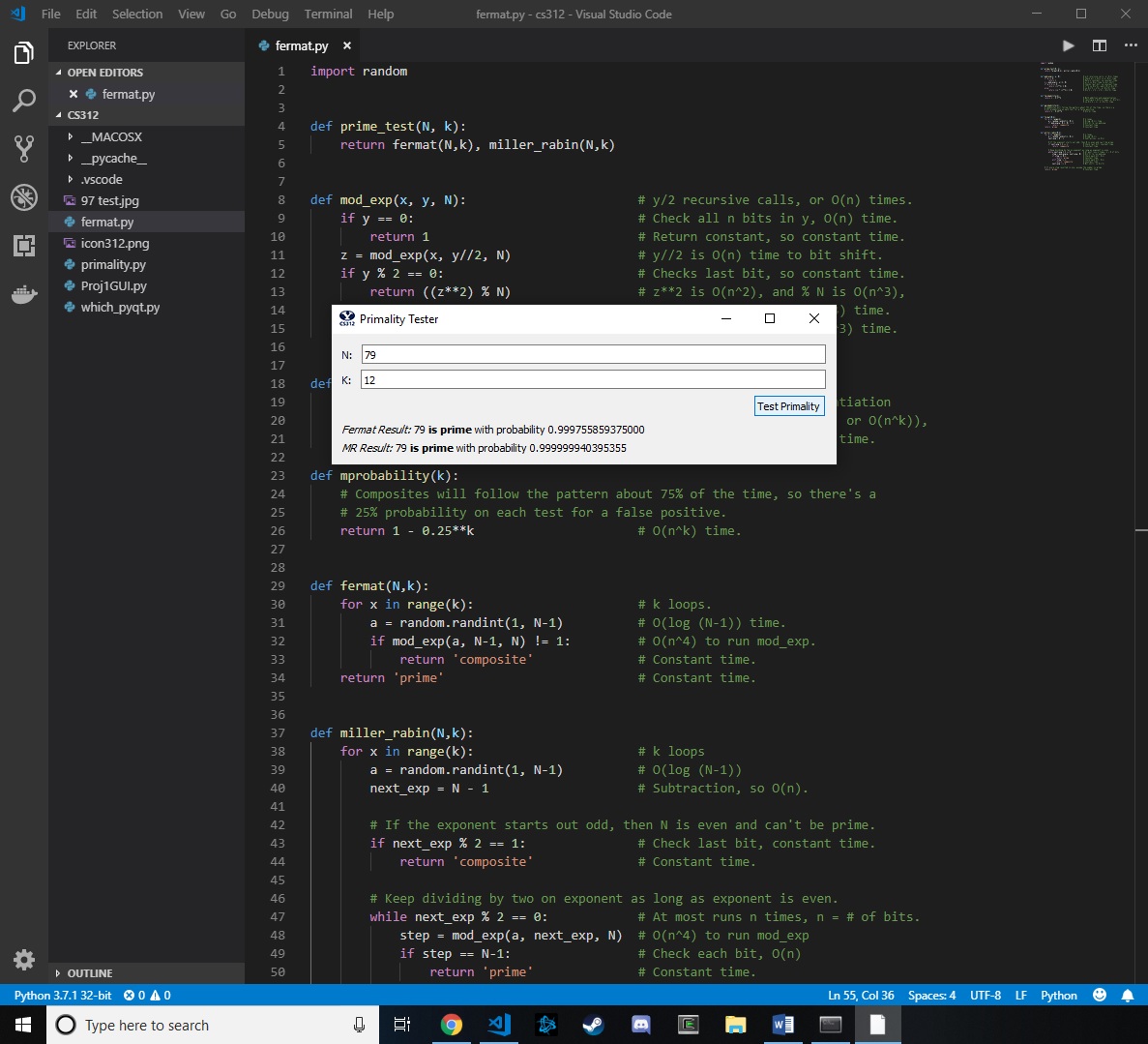
Jeff Reimschussel

Section 2

Project 1

**Screenshots**



****

**Time complexity of key portions**

mod\_exp:

There are y/2 recursive calls of function, so runs in log y, or n times through (where n is the number of bits in y).

1. Checks all n bits, so O(n) for time.
2. Return constant, so constant time.
3. O(n) time because it’s bit shifting in a parameter for the function call.
4. Simply check the last bit, so constant time.
5. The z\*\*2 is O(n^2) because it’s simply z \* z, or integer multiplication. But the modulo N takes O(n^3) time, which dominates, so O(n^3) time.
6. There are two multiplications, and one modulo, so O(n^3 + 2n^2), or O(n^3) time.

It takes O(n^3) to run O(n) times, meaning overall mod\_exp has a O(n^4) time complexity.

fermat:

There are k loops.

Runs the mod\_exp in O(n^4) time, which is the most dominant part.

Assuming k is a constant much smaller than N, overall a O(k\*n^4) or O(n^4) time complexity.

miller\_rabin:

There are k loops.

1. The while loop runs at most n times, where n = # bits of next\_exp.
2. Takes O(n^4) to run mod\_exp.
3. Checks each bit, so O(n).

The rest are checking each bit, returning a constant, or a bit shift. So O(n) for the checks, constant time for returning a constant, and O(n^2) for bit shifts.

Assuming k is a constant much smaller than N, overall a O(k\*n^4) or O(n^4) time complexity.

**Space complexity of key portions**

mod\_exp:

* O(n) space for checks.
* z, at worst, stores the value N-1, so O(n) space
* The bit shifts take O(n) space.
* Multiplications cost O(n^2) space

Has O(n) recursive calls. Overall O(n^3) space complexity.

fermat:

Has k loops. Because mod\_exp has O(n^3) space complexity, has overall a O(k\*n^3) or O(n^3) space complexity.

miller\_rabin:

Same as fermat; space complexity is dominated by mod\_exp’s O(n^3).

**Equation for calculating probabilities of fprobability & mprobability**

The Fermat theorem stated that roughly half of the numbers that followed the pattern would be Carmichael numbers. Therefore, when running *k* tests, there is a 50% chance of getting a false positive for each test. The overall accuracy after running *k* tests would be 1 - .5^*k*, where .5^*k* represents the probability that all of the values for a that suggested the number is prime were actually Carmichael numbers.

Similarly, the Miller Rabin probability states roughly 75% of numbers would not give a false positive in detecting a composite number. Therefore, the probability that each test integer *a* in *k* tests were actually Carmichael numbers is 1 - .25^*k.*

**Code**

def mod\_exp(x, y, N): # y/2 recursive calls, or O(n) times.

if y == 0: # Check all n bits in y, O(n) time.

return 1 # Return constant, so constant time.

z = mod\_exp(x, y//2, N) # y//2 is O(n) time to bit shift.

if y % 2 == 0: # Checks last bit, so constant time.

return ((z\*\*2) % N) # z\*\*2 is O(n^2), and % N is O(n^3),

else: # so O(n^3 + n^2), or O(n^3) time.

return ((x \* (z\*\*2)) % N) # O(n^2 + n^2 + n^3) = O(n^3) time.

def fprobability(k):

return 1 - 0.5\*\*k # Both addition and exponentiation

# (multiply n bits k times, O(n^k)),

# so O(n^k + n), or O(n^k) time.

def mprobability(k):

# Composites will follow the pattern about 75% of the time, so there's a

# 25% probability on each test for a false positive.

return 1 - 0.25\*\*k # O(n^k) time.

def fermat(N,k):

for x in range(k): # k loops.

a = random.randint(1, N-1) # O(log (N-1)) time.

if mod\_exp(a, N-1, N) != 1: # O(n^4) to run mod\_exp.

return 'composite' # Constant time.

return 'prime' # Constant time.

def miller\_rabin(N,k):

for x in range(k): # k loops

a = random.randint(1, N-1) # O(log (N-1))

next\_exp = N - 1 # Subtraction, so O(n).

# If the exponent starts out odd, then N is even and can't be prime.

if next\_exp % 2 == 1: # Check last bit, constant time.

return 'composite' # Constant time.

# Keep dividing by two on exponent as long as exponent is even.

while next\_exp % 2 == 0: # At most n times, n = # of bits.

step = mod\_exp(a, next\_exp, N) # O(n^4) to run mod\_exp

if step == N-1: # Check each bit, O(n)

return 'prime' # Constant time.

elif step != 1: # Check each bit, O(n)

return 'composite' # Constant time.

next\_exp //= 2 # Bit shift, so O(n^2)

# If every step resulted in one, assume the number is prime.

return 'prime' # Constant time.