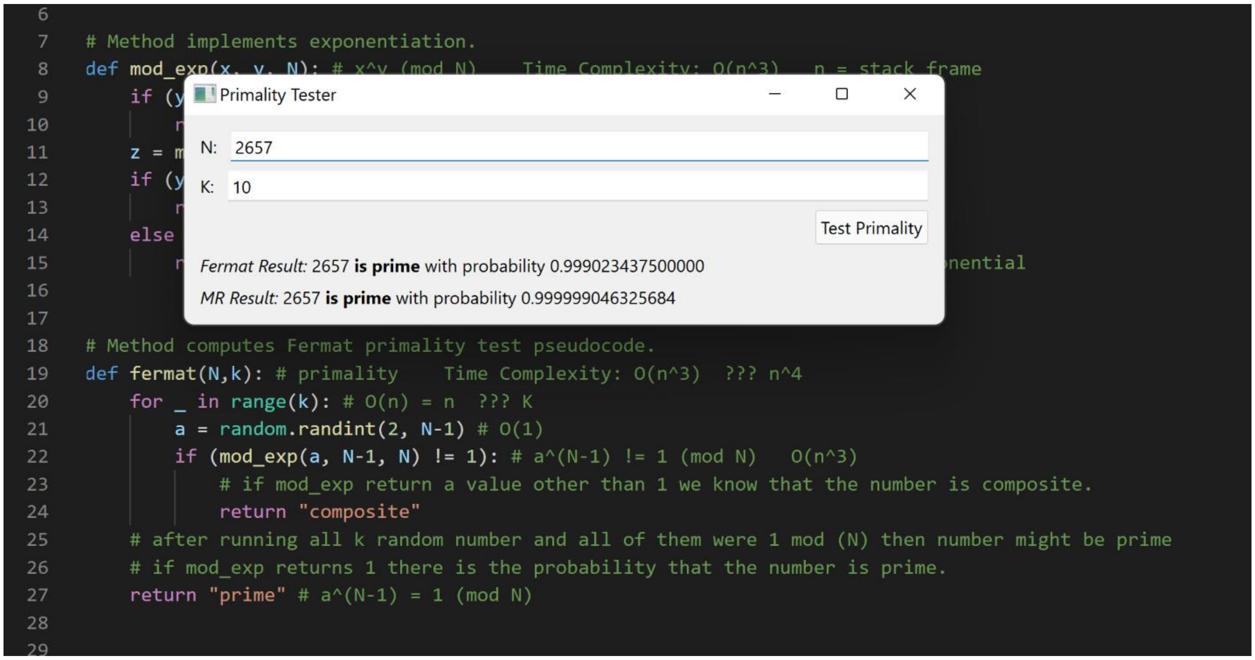
Project #1: Fermat's Primality Test

1.



2.

Code contains comments to document of what each part and method is doing Code also shows part a: A correct implementation of modular exponentiation. Part b: A correct implementation of the Fermat primality tester. And part c: A correct implementation of the Miller-Rabin algorithm.

```
import random

def prime_test(N, k):
    # This is main function, that is connected to the Test button. No need to modify.
    return fermat(N,k), miller_rabin(N,k)

# Method implements exponentiation. x^y (mod N)
def mod_exp(x, y, N): # Time Complexity: O(n^3)   n = stack frame
    if (y == 0) :
        return 1
    z = mod_exp(x, y//2, N) # y//2 = math.floor (y/2)    O(n)
    if (y % 2 == 0) : #                                  O(1)
        return (z**2) % N  #                            O(1) and O(1)
     else :
        return (x * (z**2)) % N  # O(n^2) for multiplication and O(1)

# Method computes Fermat primality test pseudocode.
def fermat(N,k): # Time Complexity: O(n^3)
    for _ in range(k): # K times
```

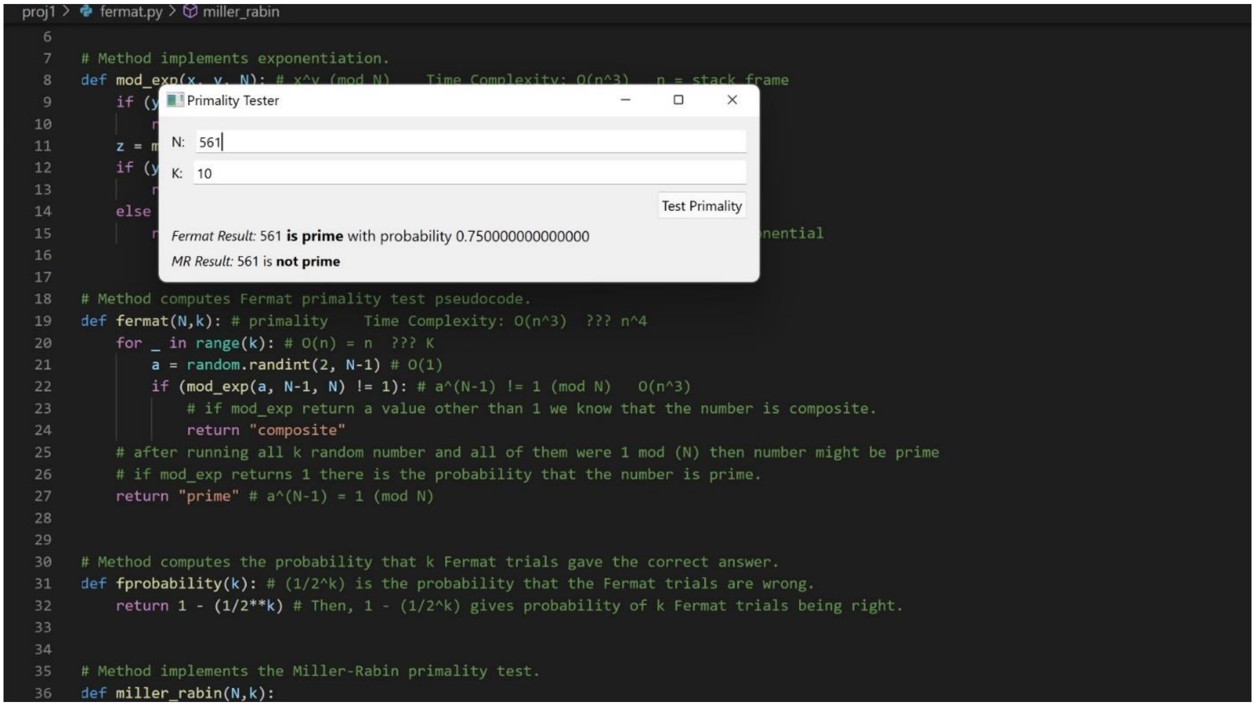
```python
        a = random.randint(2, N-1) # O(1)
        if (mod_exp(a, N-1, N) != 1): # a^(N-1) != 1 (mod N)   O(n^3)
            # if mod_exp return a value other than 1 we know that the number is composite.
            return "composite"
    # after running all k random number and all of them were 1 mod (N) then number might be prime
    # if mod_exp returns 1 there is the probability that the number is prime.
    return "prime" # a^(N-1) = 1 (mod N)


# Method computes the probability that k Fermat trials gave the correct answer.
def fprobability(k): # (1/2^k) is the probability that the Fermat trials are wrong.
    return 1 - (1/2**k) #Then, 1 - (1/2^k) gives probability of k Fermat trials being right.


# Method implements the Miller-Rabin primality test.
def miller_rabin(N,k): # Time Complexity O(n^4)
    for _ in range(k):
        a = random.randint(2, N-1)
        x = mod_exp(a, N-1, N) # a^(N-1)(mod N)   O(n^3 * n)
        if (x != 1) : return "composite"
        m = 1
        y = (N - 1)
        while(m == 1) :
            y = y//2 #floor
            m = mod_exp(a, y, N)
            if (m != 1) :
                if (m == N - 1) : break # -1 (mod N) = N-1 (mod N). If we get here, it means a passed, so we break to try another random a.
                else : return "composite"
            if (not y % 2) : break
    return "prime"

# Method computes the probability that k Miller-Rabin trials gives the correct answer.
def mprobability(k): # (1/4)^k is the probability that Miller-Rabin trial are wrong.
    return 1 - ((1/4)**k) # This gives probability of k Miller-Rabin trials is correct.
```

3.



Trying different random numbers was not very effective to find inputs for which the two algorithms disagree. I spent a good amount of time trying different numbers, but I could not find a number that made the algorithms disagree. Then, I thought of narrowing my infinite number of options. I decided to try very big numbers as well as very small numbers to see how the algorithms would respond. Then, I decided to try numbers with certain qualities like even numbers and odd numbers. After that I tried Carmichael numbers. That is how I found a couple of numbers that made the two algorithms disagree. Among those numbers was 561. Carmichael numbers can often pass Fermat's test if the test uses a relatively prime number to the Carmichael number. However, Miller-Rabin test is a more "detailed" test because a base number is tested multiple times and it checks if it is 1 or (N - 1).

4.

Modular Exponentiation

Function mod_exp(x,y,n)
Input: Two n-bit integers x and N, and integer exponent y
Output: x^y mod N
If y == 0: return 1
z = mod_exp(x, floor(y/2), N)
if y is even: return z^2 mod N
else return x * z^2 mod N

The complexity of the function is O(n^3) because there are n recursive calls of the function. We recurse as many times the bits of y can be shifted right. We have a multiplication which is O(n^2). Thus, n^2 operation n recursions give us O(n^3). The space complexity is O(n^2) because we have n recursion because and for each call we store z, which is n-bit long.

Fermat Algorithm

Function fermat (k, N)
Input: Positive integers k and N
Output: yes/no
for $a_{1,}$ a_2,..., a_k random integers
  If mod_exp(a, N-1, N) != 1: return no
return yes

The time complexity of Fermat function is (n^3). This is because the method complete k number of tests and for each completion the modular exponential method is called. This gives us O(n^3). The tie complexity of Fermat algorithm is O(n^2) because we call mod_exp method which uses space complexity of O(n^2).

Miller Rabin Algorithm

```
Function miller_rabin(k, N)
Input: Positive integers k and N
Output: yes/no
for a1, a2,..., ak random integers
   if mod_exp(a, N-1, N): return no
   while s = true
      m = mod_exp(a, floor(y/2), N)
      if(m != 1):
         if (m = N -1): s = false
         else: return no
      if(y % 2  = 1): s = false
return yes
```

Time complexity of this method is similar to Fermat's method. The difference is that we have a while loop that iterates n times where n is the bit integer of y. This gives us a time complexity of O(n^4). The space complexity is of Miller Rabin algorithm is O(n^4) because we call the mod_exp method, which uses space complexity of O(n^2), two times inside an iteration of k, which is a constant.

Probability functions

Function fprobability(k)
Input: positive integer k
Output: Fermat probability
return 1 – (1/2)^k

Time complexity of the method is O(1) since we are only doing an exponentiation. The space complexity is 0 because nothing is stored.

Function mprobability(k)
Input: positive integer k
Output: Miller-Rabin probability
return 1 – (1/4)^k

Time complexity of the method is O(1) since we are only doing an exponentiation. The space complexity is 0 because nothing is stored.

5.

From the textbook we learn that for Fermat's algorithm we have 0.5 chance that the number entered as input could be composite. If k is the number of random numbers that we try as input for the Fermat algorithm we obtain (1/2)^k. This, however, is the probability of Fermat algorithm being wrong. Thus, we can obtain the probability of the algorithm being right with the equation: 1-(1/2)^k. We also learn from the textbook that the chance that an input for the Miller-Rabin algorithm passing as a false prime is 1/4. If k is the number of random numbers used as inputs for Miller-Rabin algorithm we obtain the following. (1/4)^k. Again, this is the probability of Miller-Rabin being wrong for k random numbers. This leads us to the probability of correctness for the Miller-Rabin algorithm which is 1- (1/4)^k.