Cannon Farr – Fermat Report

Part 1: (See Appendix)

Part 2: Time and space complexity:

Modexp:

```
def mod_exp(x, y, N):
    if y == 0:
        return 1
    z = mod_exp(x, y//2, N)
    if y % 2 == 0:
        return (z*z) % N
    else:
        return (x*z*z) % N
```

The complexity of the modExp algorithm is O($\log^2 n$) The recursive algorithm runs and does a right shift on an n-bit number on every call which gives time complexity of *n.* Additionally, within each call, there is a multiplication of those n-bit numbers and we know that multiplication has a time complexity of n^2. So combining those two operations together gives us a final time complexity of **O($\log^2$n).**

The space complexity is rather simple. We store z in every recursive call. From our time calculation, we know that the function is called *n* times. So we know that there will be *zn* space needed. Since z is just the size of an n-bit number, we know we need nlog(n) space.

Fermat:

```
def fermat(N,k):
    for i in range(k):
        a = random.randint(2,N-1)
        if mod_exp(a, N-1, N) != 1:
            return 'composite'
    return 'prime'
```

The complexity of the Fermat algorithm is $k\log^2(n)$because we run the mod_exp algorithm k times in a loop. However k is a constant and so we can classify the fermat algorithm in **O($\log^2$n)** time.

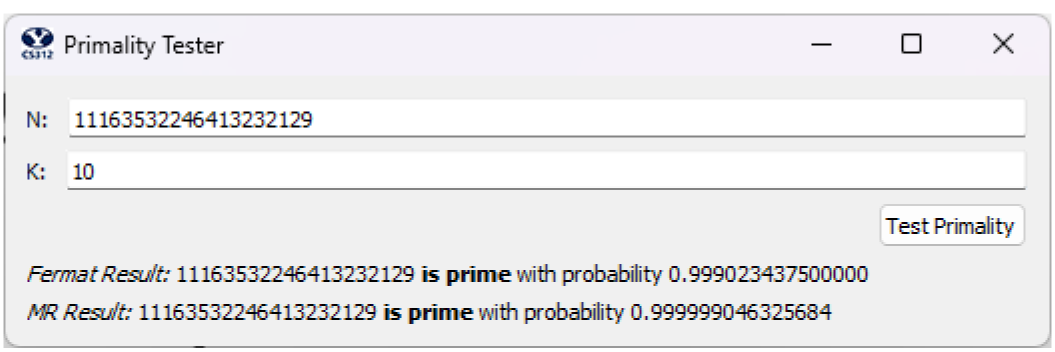
The space complexity is going to be the same as the mod_exp because in comparison to the fermat algorithm, the mod_exp space dominates the space usage. So it is nlog(n) space.

Miller-Rabin:

```python
def miller_rabin(N,k):

    sequence = []
    exp = N-1
    for i in range(k):
        a = random.randint(2,N-1)
        if mod_exp(a, N-1, N) != 1:
            return 'composite'
        print('a: ' + str(a))

        while exp % 2 == 0:
            modResult = mod_exp(a, exp, N)
            print('exp: ' + str(exp))
            print('mod: ' + str(modResult))
            if modResult == 1:
                sequence.append(1)
            elif modResult == N-1 and sequence[-1] == 1:
                return 'prime'
            else:
                return 'composite'

            exp /= 2
        return 'prime'
```

The complexity of the Miller-Rabin test adds on to the Fermat test. For each iteration of k, we will do a Fermat test which is time O($\log^2$n). Additionally, we will take the square root of the exponent (essentially halving the bits) until we can't anymore for each iteration. That additional operation is O(log(n). So in short, we are doing O($\log^2$(n)) operations for log(n) times. Which gives us a final complexity of **O($\log^3$(n)).**

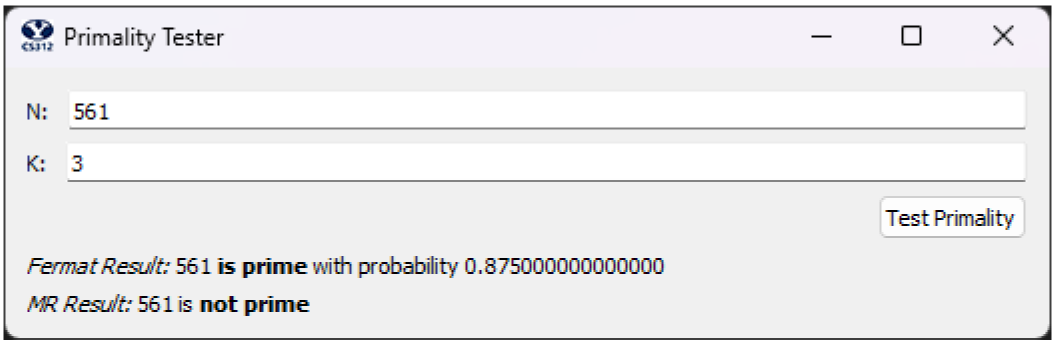
Part 3: Screenshot of working program

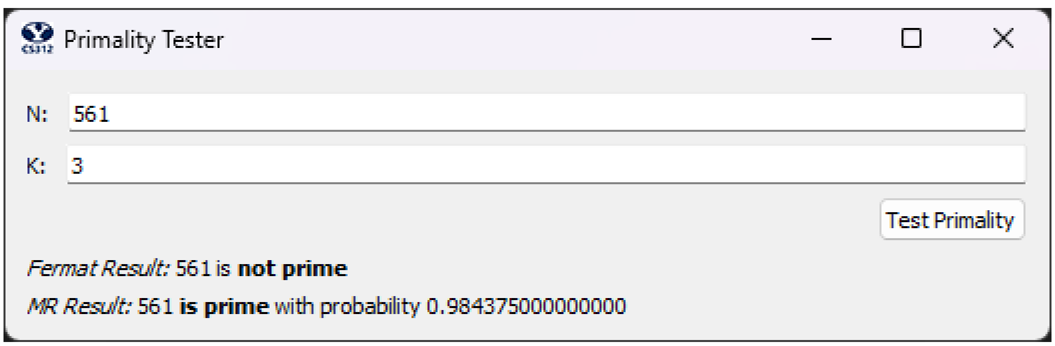


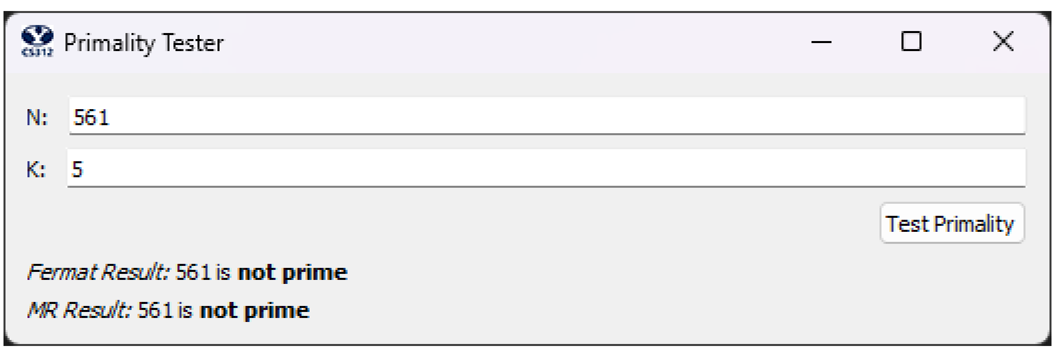
Part 4: Experimentation of inputs:

We can use Carmichael numbers to find cases where the two algorithms would disagree. Sometimes the Fermat would pass and other times the Miller Rabin would pass as shown below.

However, I also found that as you increased k, the likelihood and probability of the two algorithms agreeing goes up.







Part 5: Probability tests:

Fermat:

The probability for a singular Fermat test being correct is ½. And so for each additional iteration of the algorithm with varying numbers to test, the probability is halved. The equation we can use to find the probability is $P = 1 - (1/2)^k$

Miller Rabin:

The Miller Rabin test actually gives a higher probability of being correct with ¾. So similarly, we can reduce the error by each iteration k. This gives us an equation $P = 1 - (1/4)^k$

Appendix:

```python
import random

# This is 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, y//2, N)
    if y % 2 == 0:
        return (z*z) % N
    else:
        return (x*z*z) % N

# You will need to implement this function and change the return value.
def fprobability(k):
    return 1 - (.5 ** k)

# You will need to implement this function and change the return value.
def mprobability(k):
    return 1 - (1/(4**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 likley 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):
        a = random.randint(2,N-1)
        if mod_exp(a, 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'.
#
```

```python
# To generate random values for a, you will most likley want to use
# random.randint(low,hi) which gives a random integer between low and
#  hi, inclusive.
def miller_rabin(N,k):

    sequence = []
    exp = N-1
    for i in range(k):
        a = random.randint(2,N-1)
        if mod_exp(a, N-1, N) != 1:
            return 'composite'
        print('a: ' + str(a))

        while exp % 2 == 0:
            modResult = mod_exp(a, exp, N)
            print('exp: ' + str(exp))
            print('mod: ' + str(modResult))
            if modResult == 1:
                sequence.append(1)
            elif modResult == N-1 and sequence[-1] == 1:
                return 'prime'
            else:
                return 'composite'

            exp /= 2
        return 'prime'
```