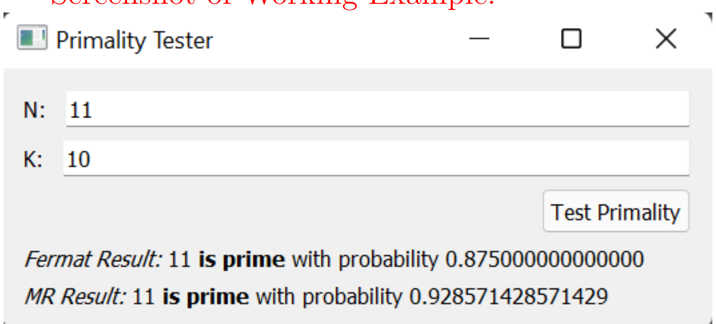
Project-1 Report

Christian Liechty

January 24, 2022

Screenshot of Working Example:



Code that was written:

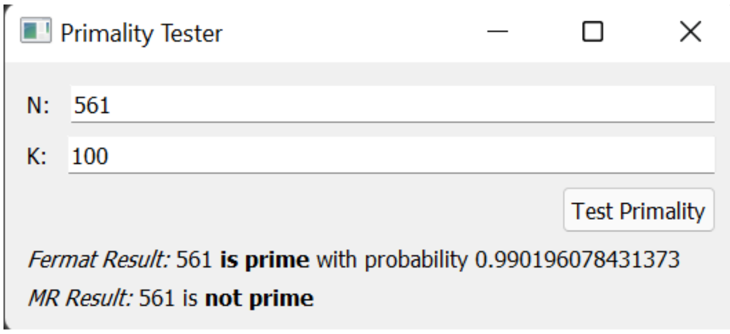
```
import random
```

```
import math


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)


def mod_exp(x, y, N):
    #Implement modular exponentiation from Figure 1.4, Pg. 19
    if y == 0:
        return 1
    """
    Every multiply and divide is O(n^2)
    The amount of multiplies/divides (or depth) is determined by y
    Therefore, O(yn^2) == O(n^3)
    """
    z = mod_exp(x, math.floor(y/2), N)
    if y % 2 == 0:
        return (z**2) % N
    else:
        return x * (z**2) % N


def fprobability(k):
    # Returns the probability that N is prime, given k trials, with Fermat
    # primality test
    fprob = 1 - (1/(2^k))
    return fprob


def mprobability(k):
    # Returns the probability that N is prime, given k trials, with
    # Miller-Rabin primality test
    mprob = 1 - (1/(4^k))
    return mprob
```

```
def fermat(N,k):
    #Implement Fermat primality test
    """
    In this algorithm, we use the modular exponentiation function that
    had a complexity of O(n^3). We use this function 'k' times in the
    for loop. Everything else takes O(1). Therefore, this primality test
    is O(kn^3) = O(n^3).
    """
    #create for loop in range of k, or trial attempts
    for i in range(k):
        #generate random value for 'a' between 2 and N-1
        a = random.randint(2, N-1)
        #equivalent to: (a ** (N-1)) % N. if this equates to 1, there is a
        # chance the number is prime
        if mod_exp(a, N-1, N) == 1:
            return 'prime'
        else:
            break

    return 'composite'


def miller_rabin(N,k):
    # Implement Miller-Rabin primality test
    # create for loop in range of k, or trial attempts
    """
    In this algorithm, we use the modular exponentiation function that
    had a complexity of O(n^3). We use this function 'k' times in the
    for loop. Everything else takes O(1). Miller-Rabin differs from Fermat
    because there is another loop and further multiplies/divides. Therefore,
    this primality test is O(kn^3kn) = O(n^4).
    """
    for i in range(k):
        # generate random value for 'a' between 2 and N-1
        a = random.randint(2, N - 1)
        # condition to check for primality
        #equivalent to: (a ** (N-1)) % N. if this equates to 1, there is
```

```
        # a chance the number is prime
        if mod_exp(a, N - 1, N) == 1:
            z = (a ** (N - 1))
            #taking sqrt of exponent and making sure it is an even value;
            # stop loop if odd
            while (z ** (1/2)) % 2 == 0:
                #further testing in order to eliminate Carmichael numbers
                if (z ** (1/2)) % N == 1 or -1:
                    return 'prime'
        else:
            break

    return 'composite'


"""
I tried many different inputs to see what would yield different results
for the two tests. I tested several large even numbers to make sure those
came back as composite. I tried many different random numbers with varying
trial size to see what the results would be.
For the most part, the tests did fairly similiar; however, the Miller-Rabin
test did much better with Carmichael numbers. I tested 561 and 1105, both
of which are Carmichael numbers. The Fermat test would give different answers,
but the Miller-Rabin was much better about consistently removing these
Carmichael numbers.
"""
```

Primality Test Disagreement:



Time and Space Complexity:

Modular Exponentiation:
Every multiply and divide is $O(n^2)$. The amount of multiplies/divides (or depth) is determined by y. Therefore, $O(yn^2) == O(n^3)$.

Fermat Primality Test:
In this algorithm, we use the modular exponentiation function that had a complexity of $O(n^3)$. We use this function 'k' times in the for loop. Everything else takes $O(1)$. Therefore, this primality test is $O(kn^3) = O(n^3)$.

Miller-Rabin Primality Test:
In this algorithm, we use the modular exponentiation function that had a

complexity of O(n^3). We use this function 'k' times in the for loop. Everything else takes O(1). Miller-Rabin differs from Fermat because there is another loop and further multiplies/divides. Therefore, this primality test is O(kn^3kn) = O(n^4).

Probabilities:

The probability of a number actually being prime with the Fermat test is: 1 - $1/2^k$.
The probability of a number actually being prime with the Miller-Rabin test is: 1 - $1/4^k$.

GitHub Link:

https://github.com/liecchr2/Project-1.git