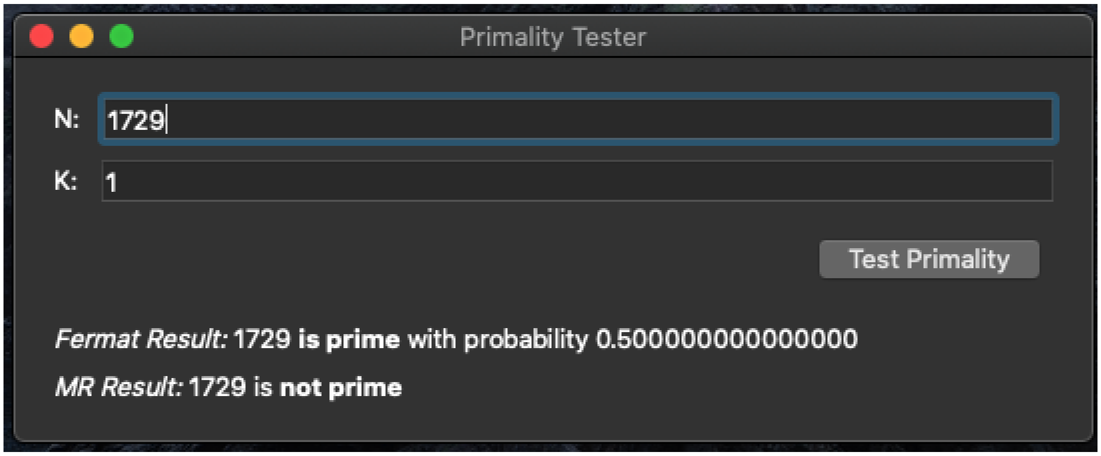
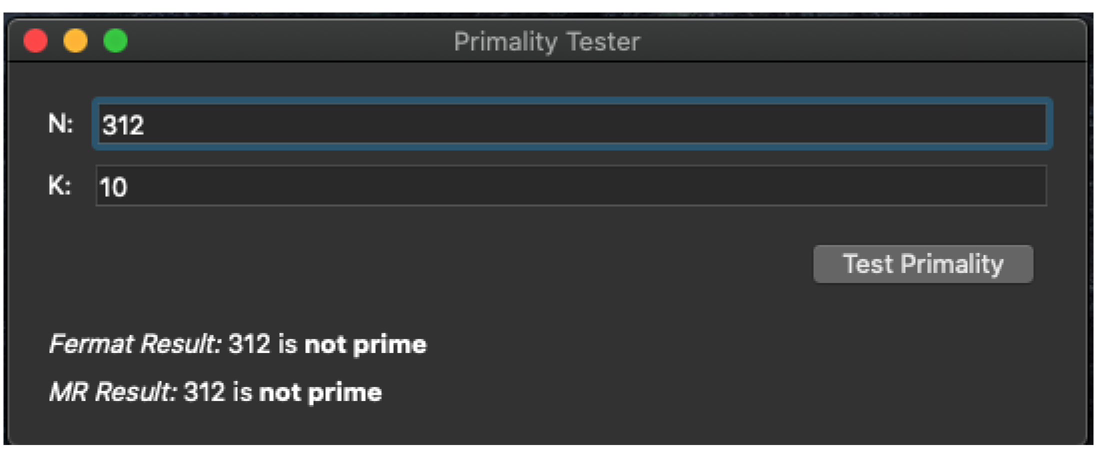
Screenshots:

Test with Carmichael number 1729 with 1 trial

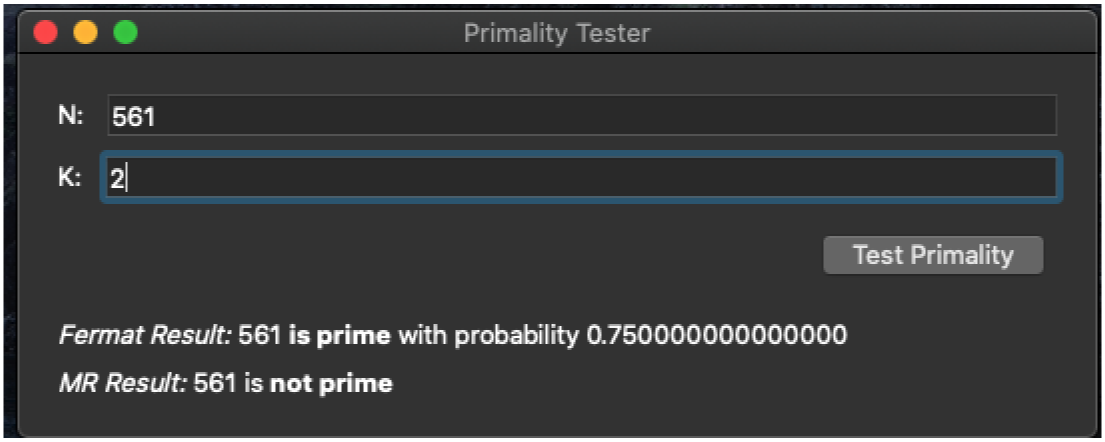


Test with Carmichael number 1729 with 10 trials

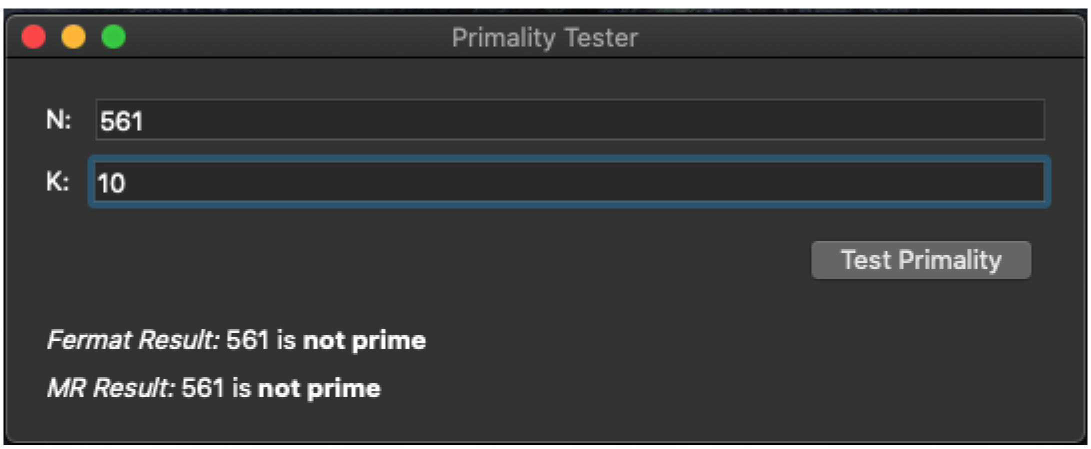
Test with number 312 with 10 trials



Test with Carmichael number 561 with 2 trials



Test with Carmichael number 561 with 10 trials



Code:

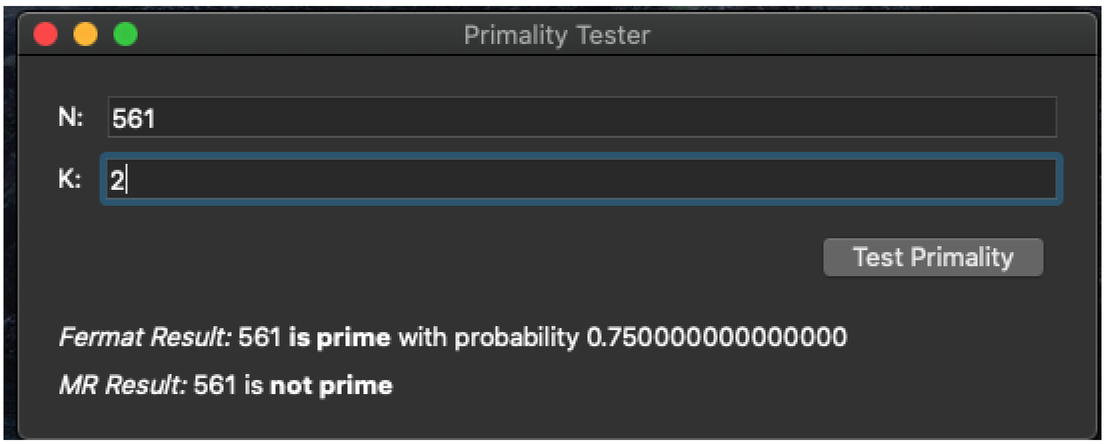
```
import random
import math

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)
```

```python
def isEven(n):  # Helper function to identify even numbers
    if n % 2 == 0:  # O(n^2) because of division in order to check for even numbers
        return True
    else:
        return False

def mod_exp(x, y, N):
    if y == 0:
        return 1  # O(1)
    z = mod_exp(x, math.floor(y/2), N)  # O(n^2)
    if isEven(y):
        return z**2 % N  # O(n^2)
    else:
        return x * (z**2) % N  # O(n^2)

    # Total running time for the function above is O(n^3)

def fprobability(k):
    # You will need to implement this function and change the return value.
    return 1 - (1 / 2**k)
    # runtime is O(n^2) - substraction is O(n),
    # division is O(n^2), exponentiation is O(n)

def mprobability(k):
    # You will need to implement this function and change the return value.
    return 1 - (1 / 4**k)
    # runtime is O(n^2) - substraction is O(n),
    # division is O(n^2), exponentiation is O(n)

def fermat(N, 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.

    low = 2
    high = N - 1
    for _ in range(0, k):
        a = random.randint(low, high)
```

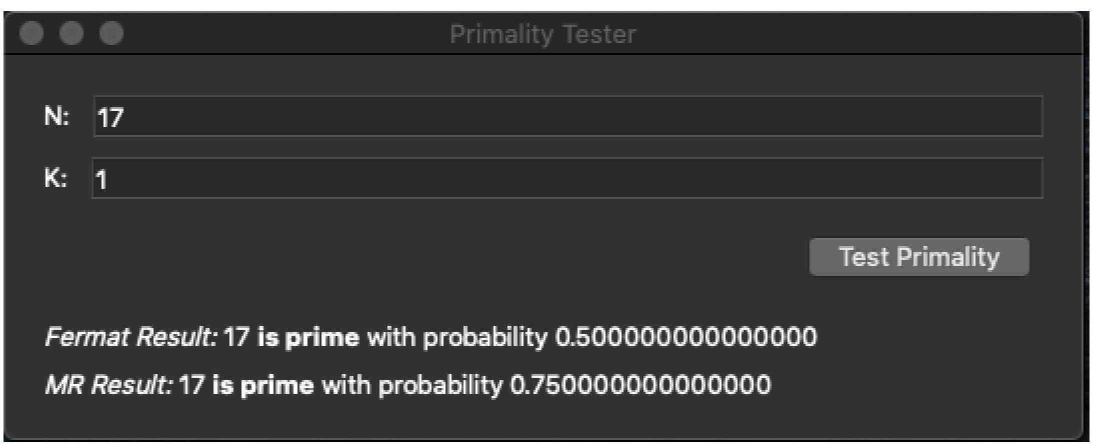
```
        if mod_exp(a, N - 1, N) == 1:  # O(n^3)
            continue
        else:
            return 'composite'  # O(1)
    return 'prime'  # O(1)
    # This function will run on O(k * n^3). O(n^3) because of the
    # mod_exp function, multiplied by k because of the range

def miller_rabin(N, 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.

    low = 2
    high = N - 1
    for _ in range(0, k):
        x = (N - 1) * 2  # O(n)
        a = random.randint(low, high)

        while x > 1 and isEven(x):
            x = x / 2
            mod = mod_exp(a, x, N)  # O(n^3)
            if mod != 1:
                if mod == N - 1:
                    break
                else:
                    return 'composite'
    return 'prime'
    # This function will run on O(k * n^4). As mod_exp runs on
    # O(n^3), after accounting for that, we also account for the divisions and
    # multiplications, getting O(n^4). Also, the main loop will range up to k, thus
    # we also multiply k into the runtime.
```

Experimentation Discussion:

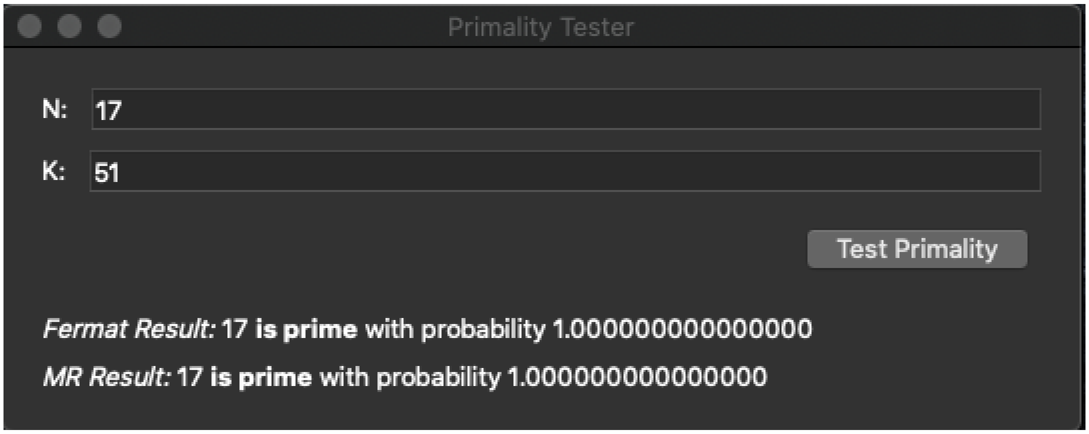
One of my experiments involved running Carmichael numbers through a different number of trials until I found a number that could be mistaken for a prime number by the Fermat primality tester. Once I found that number, I would then increase the number of tries until both tests agreed that the number was not a prime number.



I also ran some tests with a small number of tries in order to see how the probability for both algorithms differentiated.



It was really interesting to see how the probability scales as the number of tries increases, as well as reaching a point in which both algorithms reach an agreement that the number was primer after an excessive amount of tries, which for most of my times happened to be k = 51.



Time Complexity Overview:

`def mod_exp(x, y, N):` - This function runs in O(n^3). The reason for this is that the recursion and division that takes place add up to a complexity of O(n^3). Space complexity is the number of calls of n bits of input, thus totalling O(n^2)

`def fprobability(k):` - This function runs on O(n^2) because of the division that takes place in the return statement. The space complexity for this function is simply O(1).

`def mprobability(k):` - This function runs on O(n^2) because of the division that takes place in the return statement. The space complexity for this function is simply O(1).

`def fermat(N, k):` - This function runs on O(k * n^3). The reason for this is that it's calling mod_exp, which already runs on O(n^3). It also runs k times in the loop, which means that it will run depending on the number of k trials. The space complexity for this function is O(n^2).

`def miller_rabin(N, k):` - This function runs on O(k* n^4). As mod_exp runtime is O(n^3), we also account for the divisions and multiplications in the code, as well as the k range, giving O(k* n^4). The space complexity for this function is O(n^2).

Fermat vs Miller-Rabin:

It seems that some Carmichael numbers will oftentimes still pass the Fermat test. If a composite number N is not Carmichael, then the equation will detect that the compositeness is at the very least 50%. However, for the Miller-Rabin test, every composite number will be detected with a probability of at least 50%. Thus, the correctness of the probability is independent of the input N. This is what makes the correctness of the Miller-Rabin much stronger than Fermat's.