Project 1: Fermat & Miller-Rabin

Time and Space Complexity

I don't believe the space complexity really matters. We aren't maintaining any arrays, any number we need is either generated and discarded on each loop or is stored at the beginning and is accessed throughout execution.

Time complexity, however, is another story. For both Fermat and Miller-Rabin, the primary loop is executed at most k times. Each involve a call to modexp() every iteration, which is the dominant factor of each method, so really, it depends on the runtime of modexp(). Modexp() is a recursive function that runs as many times as it takes for y to go to 0. Y will be 0 after about log(n).

So, I would say the runtime for both fermat() and miller_rabin() is O(k(log(y)) where k is the desired number of tests and y = N-1. Since the -1 doesn't ultimately matter, we can say that the runtime of both ends up being O(k(log(n)).

```
def mod_exp(x, y, N):
    # You will need to implement this function and change the return value.
    if y == 0:                                          # constant time
        return 1                                        # constant time
    else:
        z = mod_exp(x, math.floor(y/2), N)              # recursive call that will run log(y) times
        if y % 2 == 0:                                  # constant time
            return (z ** 2) % N                         # constant time
        else:
            return (x * (z ** 2)) % N                   # constant time
```

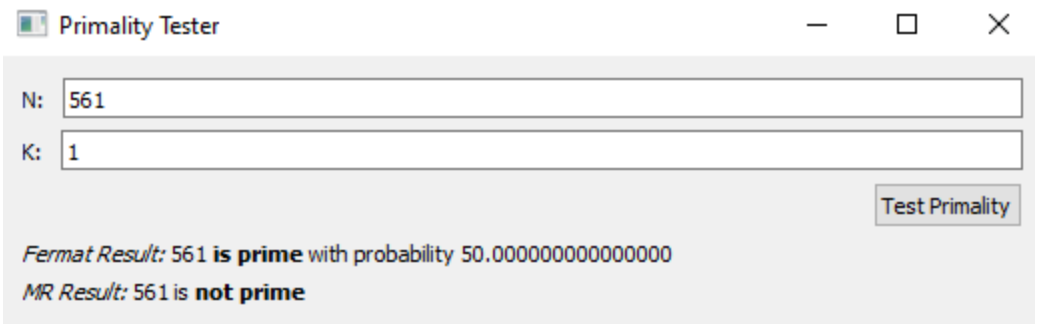
Certainty Equations

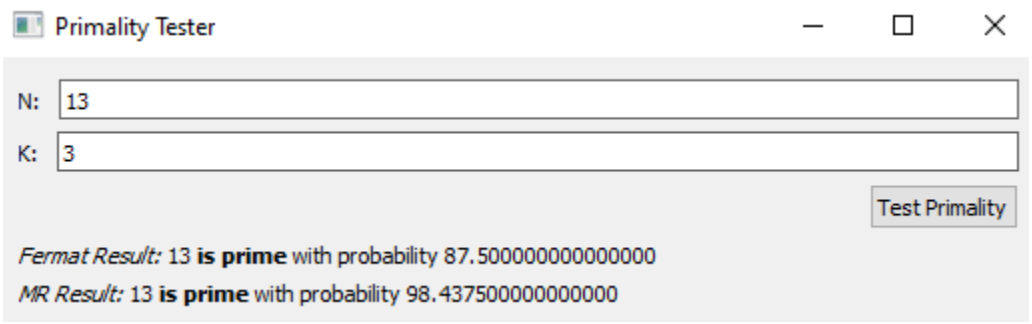
For fermat(), the equation was simple. As we discussed in class, running the test once gives you about a 50/50 shot on whether the number is actually prime. So, each successive run improves your odds by the same factor, meaning that the percentage is (1-(1/2^k)*100.

For miller_rabin(), I had to reach out to a classmate for help. I knew that it was more accurate than fermat() would be from the reading, but I wasn't sure the exact probability per run. Turns out, you need only replace the 2 with a 4. So the probability becomes 75 percent for every run, rather than 50. (1-(1/4^k)*100.

Disagreements

At first, I was confused as to why Carmichael numbers occasionally do not set off a false positive by the Fermat algorithm. I looked into it further, and found out that Carmichael numbers only set off false positives when the random number chosen is coprime with N. As shown below, fermat() determined 561 to be prime, while miller_rabin() wasn't fooled. This occurs more often with a K value of only 1. Generally, fermat() will find a number that isn't coprime with N and determine the number to be composite.


Primality Tester
N: 561
K: 1
Test Primality
Fermat Result: 561 **is prime** with probability 50.000000000000000
MR Result: 561 is **not prime**


Primality Tester
N: 13
K: 3
Test Primality
Fermat Result: 13 **is prime** with probability 87.500000000000000
MR Result: 13 **is prime** with probability 98.437500000000000