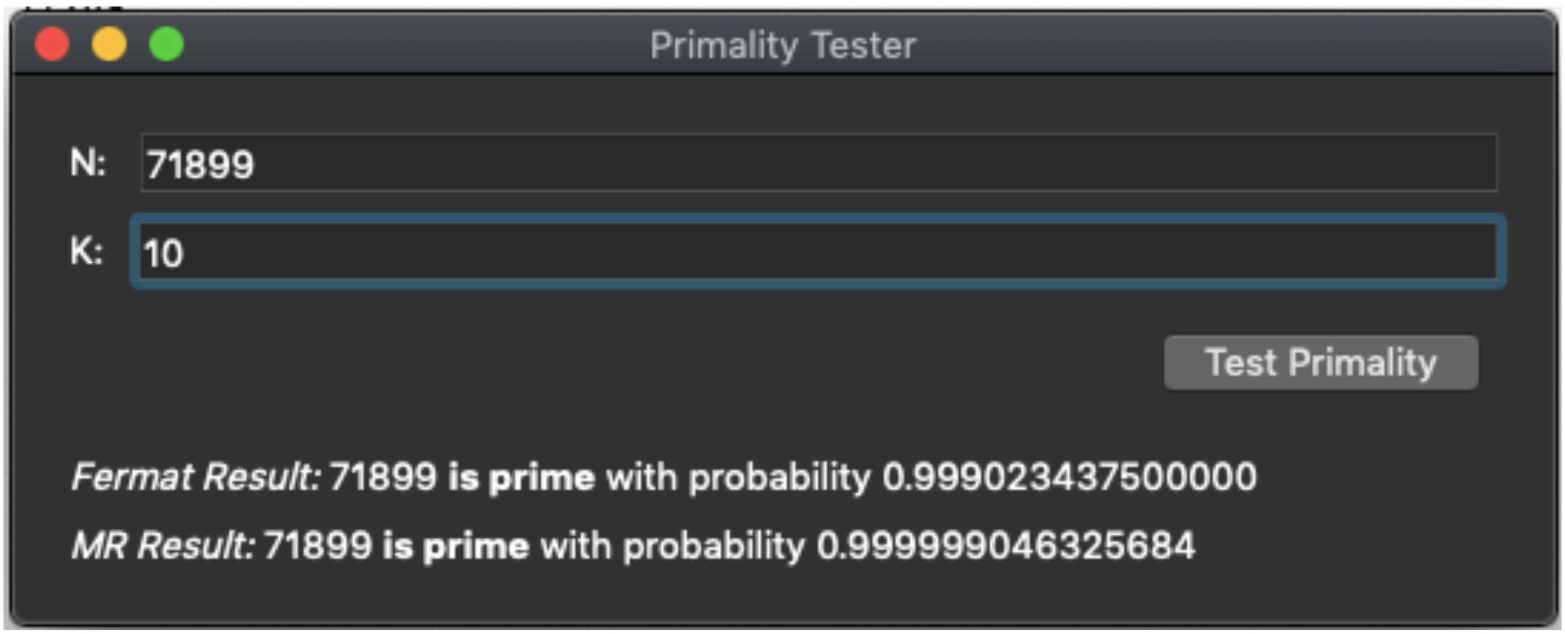
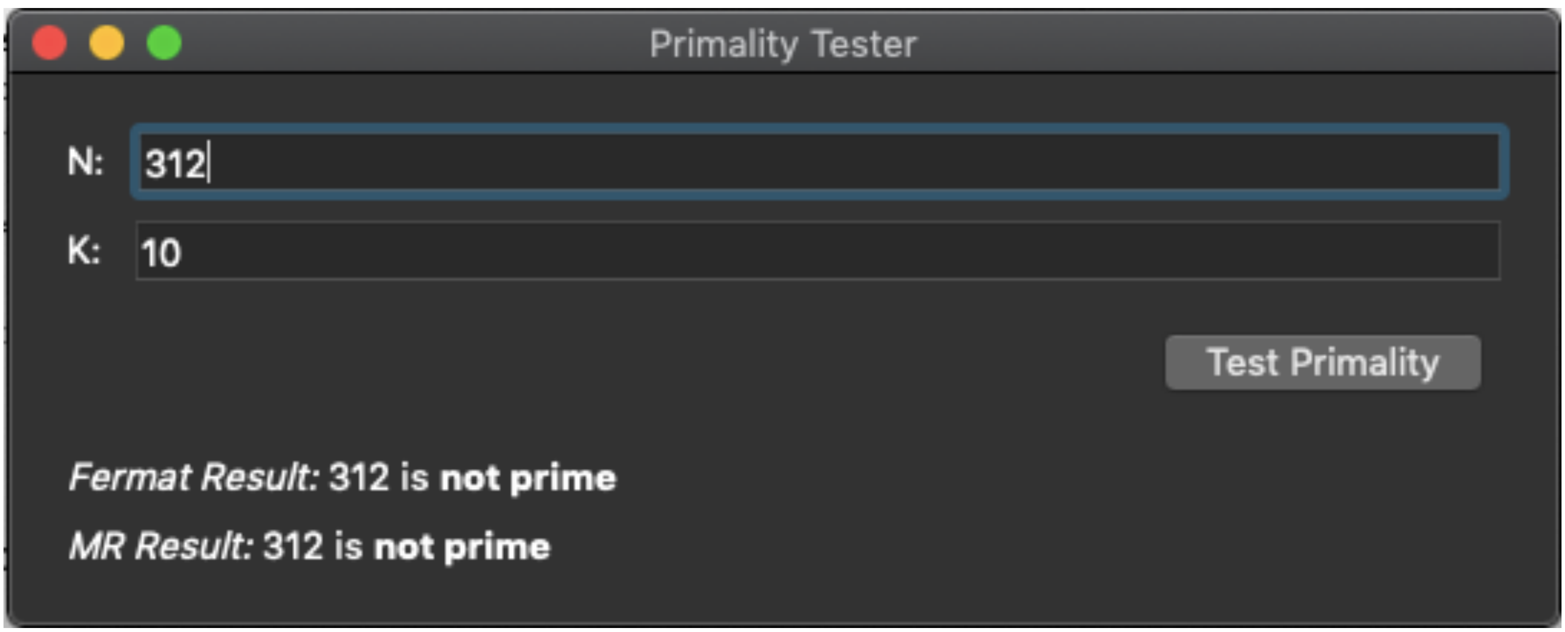
Project 1 - Primality Tester
Sam Hopkins




Primality Tester
N: 312
K: 10
Test Primality
Fermat Result: 312 is not prime
MR Result: 312 is not prime

```python
def prime_test(N, k):
    return run_fermat(N,k), run_miller_rabin(N,k)


def mod_exp(x, y, N):    # Runs in O(n^3), assuming we have n bits for input, O(n) space
    if y == 0:       # O(1) space complexity
        return 1
    z = mod_exp(x, y//2, N) # O(n) space complexity, because is recursive
    if y % 2 == 0:       # O(1), check if y is even
        return (z**2) % N    # Bit shift and then check last bit, O(n^2), space complexity O(1) y is even
    else:
        return x * (z**2) % N    # n^2 + 2n^2 so O(n^2), y is odd


# The Fermat algorithm has a 1/2 chance of being incorrect after its initial run, so
# we simply multiply that probability to the k to get the probability we are wrong after k trials,
# and subtract that from 1
def fprobability(k):     # Runs in O(n^2) time, while space complexity is O(1)
    return 1-(1/(2**k)) # Division is O(n^2), while subtraction and division are O(n) and O(2n)


# The Miller Rabin algorithm has a 1/4 chance of being incorrect after its initial run, so
# we simply multiply that probability to the k to get the probability we are wrong after k trials,
# and subtract that from 1
def mprobability(k):     # Runs in O(n^2) time, while space complexity is O(1)
    return 1-(1/(4**k)) # Division is O(n^2), while subtraction and division are O(n) and O(2n)


def run_fermat(N,k):         # Runs in O(n^3) time, but O(n) space
    for number in range(k):
        a = random.randint(2, N-1) # O(1) time, [a] becomes our random number, space O(1)
        # If mod_exp equals 1, then we know it is probability prime
        if mod_exp(a, N-1, N) == 1: # mod_exp runs in O(n^3) time, space complexity O(1)
            continue
        else:    # If mod_exp isn't 1, then we know it is composite
            return 'composite'
    return 'prime'
```

```python
def run_miller_rabin(N,k):  # Runs in O(n^4) time with space O(n)
    # We check if N is even to give quick return
    if N % 2 == 0:  # O(1) because just needs to check last bit
        return 'composite'
    for number in range(k):
        a = random.randint(1, N-1)  # O(1), [a] becomes our random number, space O(1)
        x = (N-1)   # O(n) time, set up the exponent, space O(1)
        # If x is less than 1 we are done, and if x is even then we are done
        while x > 1 and x % 2 == 0: # Repeats this at most N times, so O(n)
            if mod_exp(a,x,N) == 1:   # mod_exp runs in O(n^3) time, space O(n)
                # We take the square root, or divide the exponent by 2
                x = x/2 # O(n^2) for division
            elif mod_exp(a,x,N) == N-1: # mod_exp runs in O(n^3) time, space O(n)
                # For this a, the algorithm works, so go back and check others
                break
            else:   # It must be composite if mod_exp isn't 1 or -1
                return 'composite'
    return 'prime'
```

Time and Space complexity for my code:

prime_test(N, k):
This is the big function of the program, the one that calls both the fermat test and the miller rabin test. This function runs in total O(n^4) time with a space complexity of O(n). This comes from the space complexity of the run_miller_rabin() and the space complexity of mod_exp() when that is called.

mod_exp(x, y, N):
mod_exp() runs in O(n^3) time and O(n) space. The O(n) space comes from the recursion, creating constant variables in the first call, but then recursing up to n times. Time complexity is given from the bit shifts near the end of the function (O(n^2)), given that we have n bits for input, we reach a total big O of O(n^3) (O(n^2) * O(n)).

run_fermat(N, k):
run_fermat() runs in O(n^3) time and O(n) space. The O(n) space comes from the recursion in mod_exp(), as other than that we have constant space complexity. Time complexity comes from the recusion of mod_exp(), which as previously explained, runs in O(n^3) times. While technically run_fermat() runs in O(k * n^3) time, we can simplify this to O(n^3) because k is a constant.

run_miller_rabin(N,k):
run_miller_rabin() runs in O(n^4) time and O(n) space. The O(n) space comes from the recursion of mod_exp(), as other than that we have constant space complexity. Time complexity comes from the recursion of mod_exp(), which runs in O(n^3) as previously explained, and from the while loop, which has a time complexity of O(n). O(n^3) * O(n) = O(n^4).

Probability of p correctness:

fprobability(k):
The Fermat algorithm has a 1/2 chance of being incorrect after its initial run, so we simply exponentiate that probability to the k to get the probability we are wrong after k trials, and subtract that from 1. For example, after 12 runs, the probability that we are incorrect is 1/(2^12), so by subtracting that from 1, we get the probability that we are correct after 12 runs.

mprobability(k):
The Miller Rabin algorithm has a 1/4 chance of being incorrect after its initial run, so we simply multiply that probability to the k to get the probability we are wrong after k trials, and subtract that from 1. For example, after 12 runs, the probability that we are incorrect is 1/(4^12), so by subtracting that from 1, we get the probability that we are correct after 12 runs.