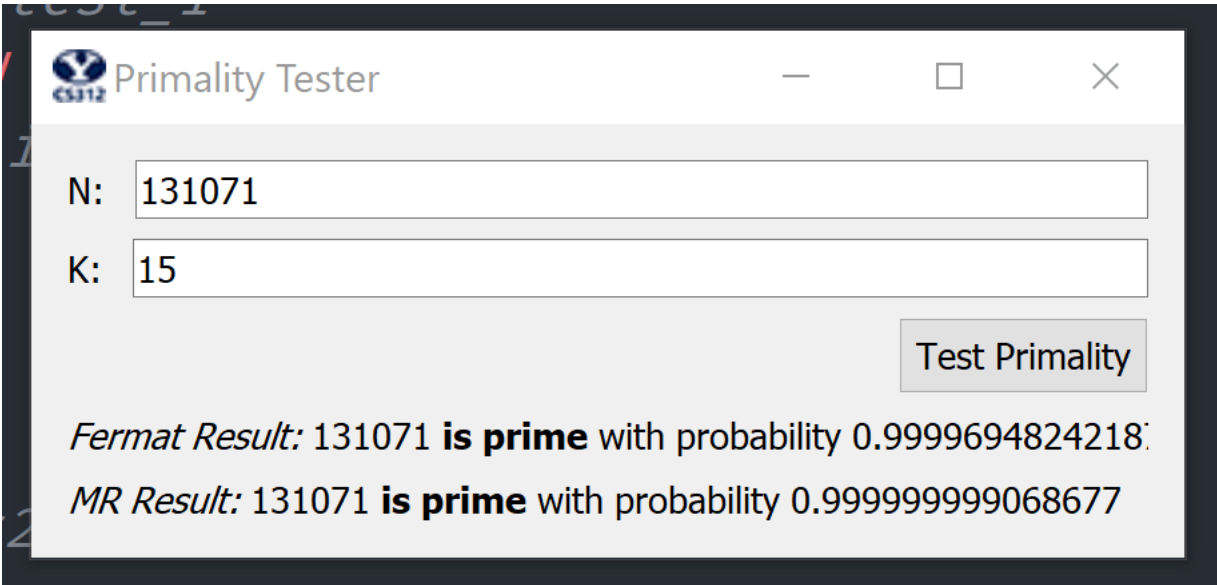


# Project 1

1.



Y CS312 Primality Tester

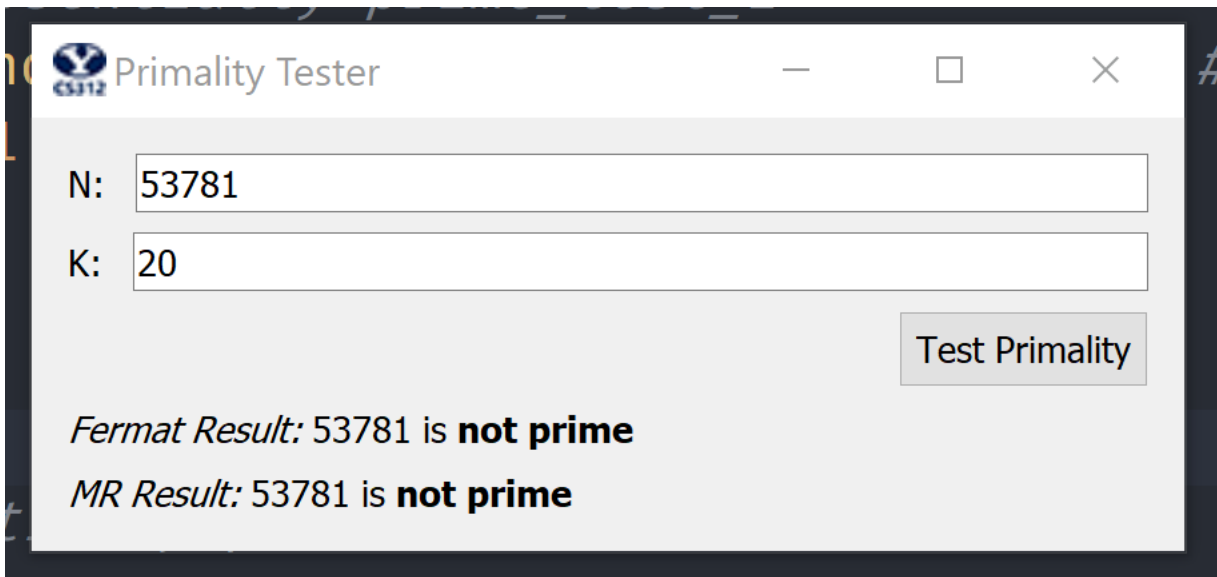
N:

K:

Test Primality

*Fermat Result:* 131071 **is prime** with probability 0.9999694824218

*MR Result:* 131071 **is prime** with probability 0.999999999068677



Y CS312 Primality Tester

N:

K:

Test Primality

*Fermat Result:* 53781 **is not prime**

*MR Result:* 53781 **is not prime**

2.

2a - 2c.

```
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):
    # 0(n^3) for function - 0(n) for run through + 0(n^2)
    if y == 0: # Base case
        return 1 # 0(1)
    # Recursive call (trunc rounds down to nearest whole number)
    z = mod_exp(x, math.trunc(y/2), N)
    if y % 2 == 0:
        return (z**2) % N # 0(n^2) + 0(1)
    return x*(z**2) % N # 0(1)
                        # 0(n^2) + 0(n^2) + 0(1)
                        # 0(n^2) + 0(n^2) + 0(n^2) + 0(1)

def fprobability(k):
    return 1 - 1/(2**k) # 0(n^3) for function - exponentiation
                        # 0(n^3)

def mprobability(k):
    return 1 - 1/(4**k) # 0(n^3) for function - exponentiation
                        # 0(n^3)

'''Singular fermat prime test called iteratively from fermat function'''
def f_prime_test(N): # Essentially prime_test_1
    # a is random number a where 1 <= a < n
    a = 1 if N == 1 else random.randint(1, N - 1) # 0(n^3) for function (mod_exp is 0(n^3))
    if mod_exp(a,N-1,N) == 1: # Fermat's little theorem
        return True # N is prime (fprobability likelihood) # 0(1)
    return False # N is composite (100%) # 0(n^3)

def fermat(N,k): # Essentially prime_test2
    for i in range(k):
        if f_prime_test(N) == False:
            return 'composite'
    return 'prime'
                        # 0(n^3) for function - 0(kn^3)+0(1)
                        # k - 0(1)
                        # 0(n^3)
                        # 0(1)
                        # 0(1)

'''Singular Millar-Rabin prime test called iteratively from miller_rabin function'''
def m_prime_test(N): # Tests one number at a time
    # a is random number a where 1 <= a < n
    a = 1 if N == 1 else random.randint(1, N - 1)
    exponent = N - 1
    while not exponent & 1:
        exponent >>= 1 # Right bit shift 1
    if mod_exp(a,exponent,N) == 1:
        return True # N is prime (mprobability likelihood)
    while exponent < N - 1:
        if mod_exp(a,exponent,N) == N - 1:
            return True # N is prime (mprobability likelihood)
        exponent <= 1 # Left bit shift 1
    return False # N is composite (100%)

def miller_rabin(N,k):
    for i in range(k):
        if m_prime_test(N) == False:
            return 'composite'
    return 'prime'
                        # 0((log n)^3) for function - calls m_prime_test which is 0((log n)
                        # k - 0(1)
                        # 0((log n)^3)
                        # 0(1)
                        # 0(1)

```

## 2d.

I started with  $N = 1$  and ascending up until  $N = 30$  with a  $K$  of 10, testing to see if all the results were correct, both for Fermat and for Miller-Rabin. All my tests passed for these. Once I felt confident that my algorithms were working for smaller number, I looked up large prime numbers on the internet and tested them, then composite numbers one away from them. All tests passed. The last set of tests I did were with Carmichael numbers. I entered them in, and with a  $k$  of 10, they mostly passed, with some exceptions. When I lowered  $K$  to one, it failed far more often for the Fermat test (meanwhile the Millar-Rabin kept passing since it handles carmichael numbers). The reason Fermat was falsely saying it was prime was because it passed on a few cases where it looked like the mod came out to be 1, when in reality that only worked on a few cases, but not all of them. A higher  $k$  makes this evident, or Millar-Rabin's test.

### 3.

Detailed throughout the code, but time and space complexity of each function also included here for convenience.

#### **mod\_exp**

$$O(n^3)$$

$O(n)$  for the run through the function +  $O(n^2)$  for the constant amount of advanced operations (multiplication, module, exponentiation)

Technically the complexity is  $O(n^2m)$ , but assuming the length of x is comparable to N, it's  $O(n^3)$

#### **fprobability**

$$O(n^3)$$

Exponentiation is  $O(n^3)$ , and the return value has to be exponentiated (k is in the exponent)

#### **mprobability**

$$O(n^3)$$

Exponentiation is  $O(n^3)$ , and the return value has to be exponentiated (k is in the exponent)

#### **f\_prime\_test**

$$O(n^3)$$

Calls `mod_exp` which is dominant with a complexity of  $O(n^3)$

$$O(n^3)$$

## fermat

Calls `f_prime_test` which is dominant with a complexity of  $O(n^3)$  (it calls `mod_exp` that has that complexity, then the for loop is negligible because it's just with a constant  $k$ ).

## m\_prime\_test

$$O(\log^3 n)$$

There are lines which are  $O(\log n)$  or the while loops, then they are combined with the two calls to `mod_exp`, therefore the overall complexity is  $O(n^3)$

## miller\_rabin

$$O(\log^3 n)$$

Calls `m_prime_test` which is dominant with a complexity of  $O(n^3)$ . Even though it has a for loop, it's negligible because we're just looping through  $k$ , a constant.



The space complexity for all the functions above is  $O(k)$   
(They don't take up extra space as they grow, they just re-use the current data structures)

## 4.

### Fermat Success Probability:

$$p = 1 - \frac{1}{2^k}$$

The success probability is just the inverse of the error:  $1 - \text{error}(\frac{1}{2^k})$ . The reason the algorithm isn't correctly 100% of the time is there can be one-sided error. For example, with Carmichael numbers, they pass enough of Fermat's Little Theorem tests that we think they are prime, when really they are not. If we up the volume of  $k$ , we'll reduce the likelihood of incorrect results. Therefore, increasing  $k$  will increase the denominator ( $2^k$ ), thus making a much smaller number (therefore significantly less percent error).

### Miller Rabin Success Probability:

$$p = 1 - \frac{1}{4^k}$$

The success probability is just the inverse of the error:  $1 - \text{error}(\frac{1}{4^k})$ . While this algorithm makes great strides in making the Fermat tests even more accurate and even builds on top of them, there can still be a level of uncertainty, depending on the number of tests ( $k$ ) we decide to do. With each iteration, more tests are made with Miller Rabin than Fermat, which is why it's  $4^k$  instead of  $2^k$ . Like with Fermat, if we up the number for  $k$ , that will increase our denominator  $4^k$ , thus making a much smaller number (therefore significantly less percent error).