



Primality Tester

N: 104729

K: 10

Test Primality

Fermat Result: 104729 **is prime** with probability 0.999023437500000

MR Result: 104729 **is prime** with probability 0.999999046325684

```
def mod_exp(x, y, N):  
    # Time complexity of this function is  $n^3$   
    # n is the length of N in bits  
    #  $O(n)$  from the recursive call  
    #  $O(n^2)$  from the multiplication  
    #  $O(n^2)$  from the %  
    #  $O(n*(n^2+n^2)) \Rightarrow O(n^3)$  overall  
    #  
    # Space complexity of this function is  $n^2$   
    # n is the length of N in bits  
    #  $n^2$  comes from having to store the data during the recursion calls  
  
    if y == 0:  
        return 1  
    z = mod_exp(x, math.floor(y/2), N)  
    if y % 2 == 0:  
        temp = z * z  
        return temp % N  
    else:  
        temp = x * z * z  
        return temp % N
```

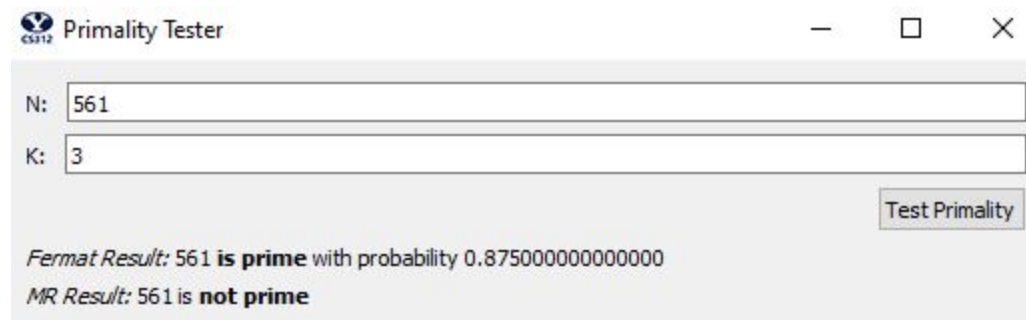
```
def fermat(N, k):  
    # Time complexity of this function is  $k*(n^3)$   
    # n is the length of N in bits  
    # k is the number of times it goes through the for loop  
    #  $O(k)$  from the for loop  
    #  $O(n^3)$  from the mod_exp function  
    #  $O(k*(n^3))$  overall  
    #  
    # Space complexity of this function is  $n^2$   
    # n is the length of N in bits  
    #  $n^2$  comes from having to store the data during mod_exp  
  
    y = int(N-1)  
  
    for i in range(k):  
        x = int(random.randint(1, y))  
        if mod_exp(x, y, N) == 1:  
            continue  
        else:  
            return 'composite'  
  
    return 'prime'
```

```

def miller_rabin(N, k):
    # Time complexity of this function is k*(n^4)
    # n is the length of N in bits
    # k is the number of times it goes through the for loop
    # Space complexity of this function is n^2
    # n^2 comes from having to store the data during mod_exp

    y = int(N - 1)
    temp = y
    for (i) in range(k):
        y = temp
        helper = 0
        x = int(random.randint(2, N-1))
        if mod_exp(x, y, N) == 1:
            while (y % 2) == 0:
                y = y/2
                if mod_exp(x, y, N) == 1:
                    continue
                else:
                    if mod_exp(x, y, N) == N-1:
                        helper = 1
                    else:
                        if helper == 1:
                            continue
                        else:
                            return 'composite'
            else:
                return 'composite'
    return 'prime'

```



CS112 Primality Tester

N: 561

K: 3

Test Primality

Fermat Result: 561 is **prime** with probability 0.8750000000000000

MR Result: 561 is **not prime**

A brief discussion of some experimentation you did to identify inputs for which the two algorithms disagree and why they do so. Include a screenshot showing a case of disagreement:

The two algorithms have a chance to disagree when the number that is being checked for primality is a Carmichael Number. Carmichael numbers fool the Fermat test for all values relatively prime to the chosen N value. The Rabin Miller test makes it so that we can tell if a number is prime or not, even Carmichael numbers, without being fooled. I tested multiple Carmichael numbers and as long as I put a relatively low K number down, after clicking "Test Primality" a couple times, the two algorithms would disagree with each other.

Explain the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your code:

Mod_Exp:

Time Complexity:

- n is the length of N in bits
- $O(n)$ from the recursive call
- $O(n^2)$ from the multiplication
- $O(n^2)$ from the $\%$
- $O(n \cdot (n^2 + n^2)) \Rightarrow \mathbf{O(n^3)}$ overall

Space Complexity:

- n is the length of N in bits
- $O(n^2)$ comes from having to store the data during the recursion calls
- $\mathbf{O(n^2)}$ overall

Fermat:

Time Complexity:

- n is the length of N in bits
- k is the number of times it goes through the for loop
- $O(k)$ from the for loop
- $O(n^3)$ from the mod_exp function call
- $\mathbf{O(k \cdot (n^3))}$ overall

Space Complexity:

- n is the length of N in bits
- k is the number of times through the for loop
- $O(k)$ from the for loop
- $O(n^2)$ comes from having to store the data during mod_exp
- $O(k + (n^2)) \Rightarrow \mathbf{O(n^2)}$ overall

Miller-Rabin:

Time Complexity:

- n is the length of N in bits
- k is the number of times it goes through the for loop
- $O(k)$ from the for loop
- $O(n)$ from the while loop
- $O(n^3)$ from the mod_exp function call
- $\mathbf{O(k \cdot (n^4))}$ overall

Space Complexity:

- n is the length of N in bits
- k is the number of times through the for loop
- $O(k)$ from the for loop
- $O(n^2)$ comes from having to store the data during mod_exp
- $O(k + (n^2)) \Rightarrow \mathbf{O(n^2)}$ overall

```

def fprobability(k):
    # You will need to implement this function and change the return value.
    return 1-(1/pow(2, k))

def mprobability(k):
    # You will need to implement this function and change the return value.
    return 1-(1/pow(4, k))

```

Discuss the two equations you used to compute the probabilities p of correctness for the two algorithms (Fermat and Miller-Rabin):

Fermat:

The probability of picking a value that doesn't pass the primality test is at first $\frac{1}{2}$. However, the chance of picking two in a row that fail is $\frac{1}{2} * \frac{1}{2} = \frac{1}{4}$. Three in a row would be $\frac{1}{2} * \frac{1}{2} * \frac{1}{2} = \frac{1}{8}$. The probability of picking all numbers that pass the test without actually being a prime is $1/(2^k)$ where k is how many numbers are tested. $1-(1/(2^k))$ is the probability that the test said a number is prime when it truly is a prime.

Miller-Rabin:

The probability of picking a value that doesn't pass the primality test is at first $\frac{1}{4}$ (even Carmichael numbers). However, the chance of picking two in a row that fail is $\frac{1}{4} * \frac{1}{4} = \frac{1}{16}$. Three in a row would be $\frac{1}{4} * \frac{1}{4} * \frac{1}{4} = \frac{1}{64}$. The probability of picking all numbers that pass the test without actually being a prime is $1/(4^k)$ where k is how many numbers are tested. $1-(1/(4^k))$ is the probability that the test said a number is prime when it truly is a prime.