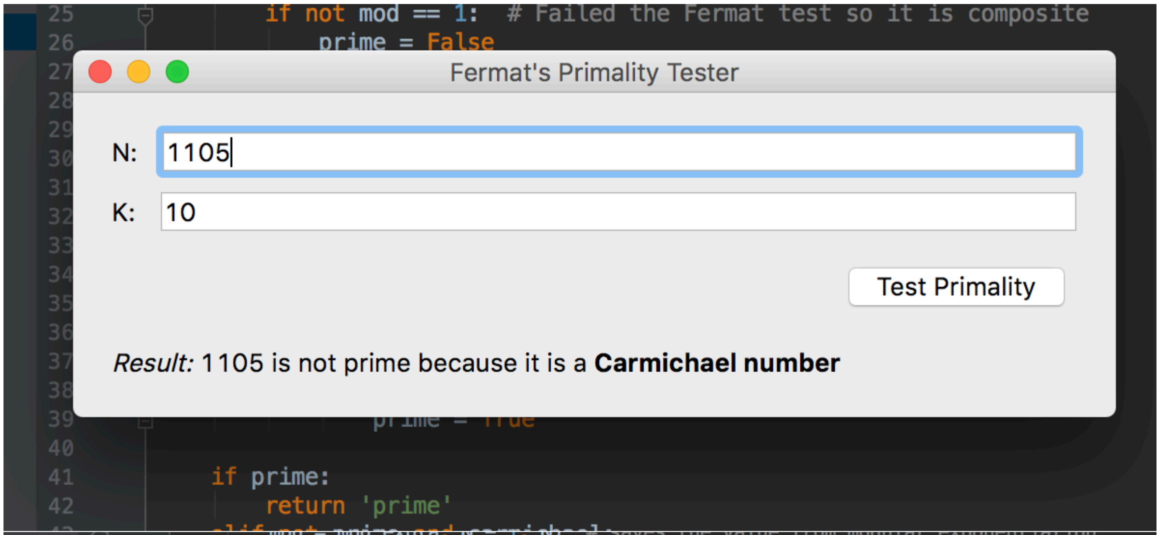
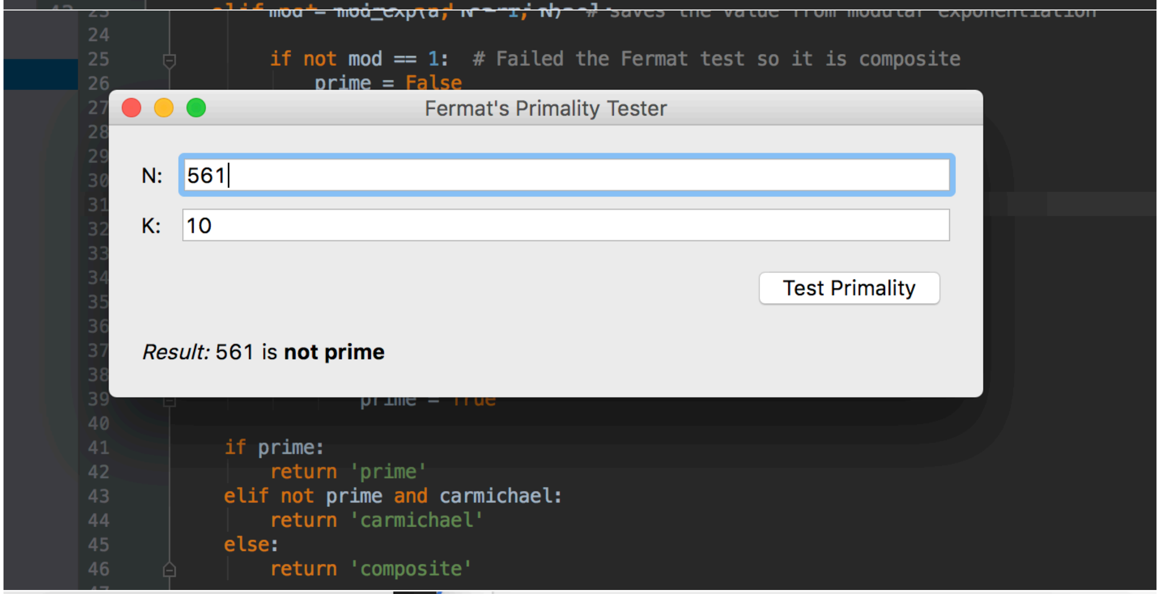
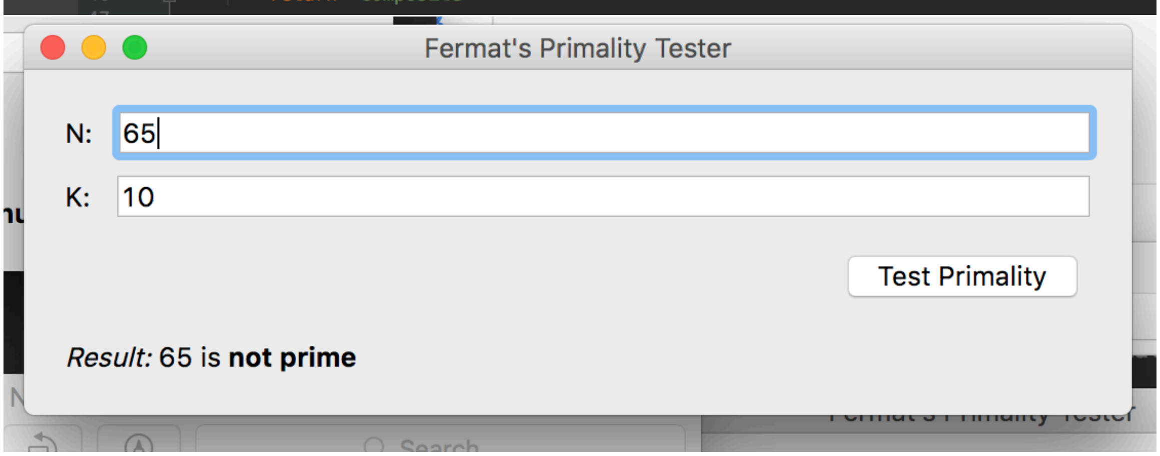
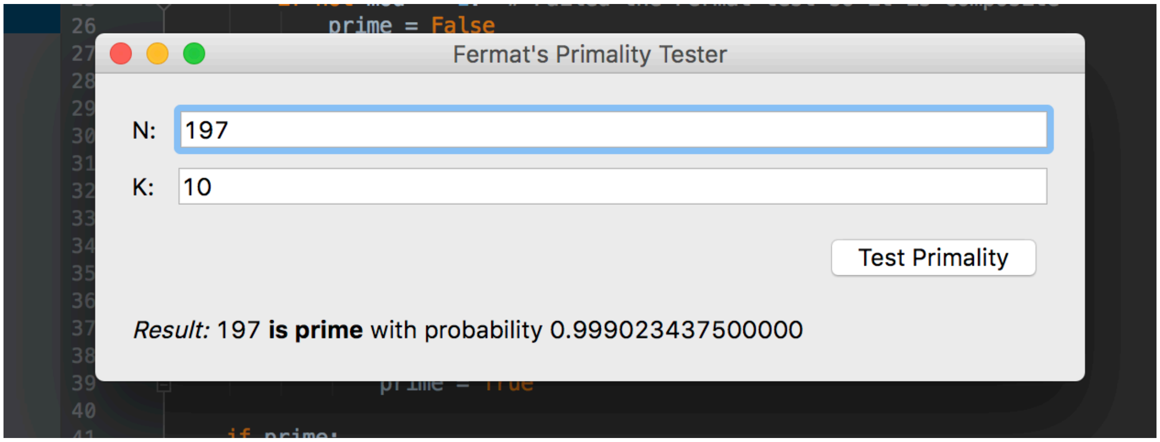

if not mod == 1: # Failed the Fermat test so it is composite
prime = False
Fermat's Primality Tester
N: 1105
K: 10
Test Primality
Result: 1105 is not prime because it is a Carmichael number
if prime:
return 'prime'


if not mod == 1: # Failed the Fermat test so it is composite
prime = False
Fermat's Primality Tester
N: 561
K: 10
Test Primality
Result: 561 is not prime
if prime:
return 'prime'
elif not prime and carmichael:
return 'carmichael'
else:
return 'composite'


Fermat's Primality Tester
N: 65
K: 10
Test Primality
Result: 65 is not prime



The above pictures show the correctness of my program. The first one shows how it can recognize a Carmichael number while the second shows a Carmichael number failing the Fermat test. Finally, the last two show how a composite number and a prime number are correctly tested.

```
import random
import math


def prime_test(N, k):

    # Has a time complexity of O(n^5)
    # Has a space complexity of O(n) since it calls mod_exp and is_carmichael

    prime = None  # Bool to keep track if prime
    carmichael = None  # Bool to keep track if carmichael number

    for i in range(1, k):   # Loop that does k random trials
        a = random.randint(1, N - 1)  # Creates a random number that is assigned to a for Fermat's Test
        mod = mod_exp(a, N - 1, N)  # saves the value from modular exponentiation

        if not mod == 1:  # Failed the Fermat test so it is composite
            prime = False
            break
        else:
            prime = True

    if prime:  # May be prime or carmichael number so need to check k times again
        for i in range(1, k):
            a = random.randint(1, N - 1)
            carmichael = is_carmichael(N, a)
            if carmichael:  # number is a carmichael number so break loop
                prime = False
                break
            else:
                prime = True

    if prime:
        return 'prime'
    elif not prime and carmichael:
        return 'carmichael'
    else:
        return 'composite'
```

```python
def mod_exp(x, y, N):

    # Has a time O(n^3) n is the number of bits in x, y or N whichever is biggest
    # Space O(n) cause each recursive call of n is stored on the stack and then deleted when returned with each call

    if y == 0:  # If the exponent is 0 from flooring it stop
        return 1
    z = mod_exp(x, math.floor(y / 2), N)  # Recurse until you get y = 0
    if y % 2 == 0:  # if y is even
        return pow(z, 2) % N
    else:  # if y is odd
        return x * pow(z, 2) % N


def probability(k):

    # Has space complexity of this function is O(n) where n is the bit size of k
    # cause as k gets larger more space is needed to store the float
    # Has time complexity of O(k)

    return 1 - (1 / math.pow(2, k))  # calculates the probability that the number is prime


def is_carmichael(N, a):
    # The time complexity here is O(n^4) since it is O(n) and calls mod_exp which is O(n^3)
    # n is the bit size of y which is the number N - 1
    # Space complexity here is O(n) since it calls mod_exp
    y = N - 1
    while True:
        if y == 1:   # This breaks if y gets to one since 1 is odd
            break
        mod = mod_exp(a, y, N)
        if not mod == 1:  # if mod_exp is not one check if its prime or carmichael
            if mod == N - 1:  # This means it is prime
                return False
            else:  # This means it is a Carmichael number
                return True
        if not (y % 2) == 0:  # Checks to see if the exponent is divisible by two if not stop
            break
        y = y / 2  # Divide exponent by 2 which is the same as sqrt the whole equation
    return False
```

My code has many subsections that call other functions so I will start with the function that doesn't call anything else mod_exp.

<u>Mod_exp Function</u>

Mod_exp takes in an x, y, and N as inputs. X is a random number raised to the y power. N is the number we are testing to see if it is prime. The function is recursive and stops once the exponent is equal to y. Y is halved with each call to mod_exp. Y is stored as an n-bit number and so halving is just a right shift. It will get to zero once it is shifted all the way to the right a O(n). The most time it will then take is when y is an odd number. This requires z and x to be multiplied together. They are both n-bits long so multiplying them is O(n^2). ***This is done n times so the overall time complexity is O(n^3) where n is the size in bits of x,y, and N (which ever is the largest of the three).***

The space complexity of mod_exp is O(n) since each recursive call creates a new runtime stack with memory for all of the local variables of that recursive call. The n in Big-O is the n-bit number y.

Probability Function

The probability function receives k as its input. K is the number of random trails performed.

My probability function has a time complexity of O(k). This is because the function calculates 2^k. This in essence is a left bit shift k times of the number 2. The space complexity is O(n) where is the bit size of k. This is because as k gets larger more space is needed to store the float with more precision.

Is_Carmichael Function

The is_carmichael receives N and a as its inputs. A is a random number chosen and N is the number we are testing if it is prime. Y is calculated right away as N – 1. This y is the exponent that raises a to a certain number. The while loop breaks away once y is 1 since it is odd and can't be used in this test, or when mod_exp returns a number that is not 1.

The time complexity here is O(n^4). This is because the while loop is called n times where n is the bit size of y. Y is right bit shifted for every call of the loop. It technically goes till n – 1 since it breaks at y equal to one but 1 is a constant and can be dropped. However, it calls mod_exp n times so since mod_exp was O(n^3) is_carmichael is O(n^4).

The space complexity here is O(n) since the while loop uses constant memory but calls mod_exp which had a space complexity of O(n) see reason above.

Prime_test function

The prime_test function receives N and k as its inputs. N is the number we are testing if it is prime and k is at most the amount of times we will run random tests. There are two for loops within the function so they both need to be looked at to see which one runs the longest to find the overall time complexity of this function. The first loop performs the Fermat test. This loop runs n times where n is equal to the number k. It calls mod_exp and as can be seen above mod_exp is O(n^3) so the first for loop is O(n^4). The second loop runs n times where n is equal to the number k again but this time it calls is_carmichael which is O(n^4), as seen above, so this loop is O(n^5). ***So, the overall time complexity of prime_test is O(n^5) since this is the worst case scenario.***

The space complexity is O(n) since it calls mod_exp and is_carmichael and these are both O(n) as seen above.

The equation I used to compute the probability p of correctness was 1 – (1 / 2^k). This takes into account the amount of random trials we used. The more trials we used the smaller 1 / 2^k is and thus overall the probability is closer to one, i.e. 100% that the number N is a prime. The closer to one the probability is the more likely that the number that came back from the tests as prime truly is prime. This is because when using Fermat's algorithm for only one 'a' there is a probability that the number N being returned as prime will be a false positive at most around 50% of the time. Thus as more trials are used the probability that N will be returned as prime when it isn't is 1 / 2^k. The more trials used the less likely the number N having a false negative exponentially drops. This is why I used the equation I used.