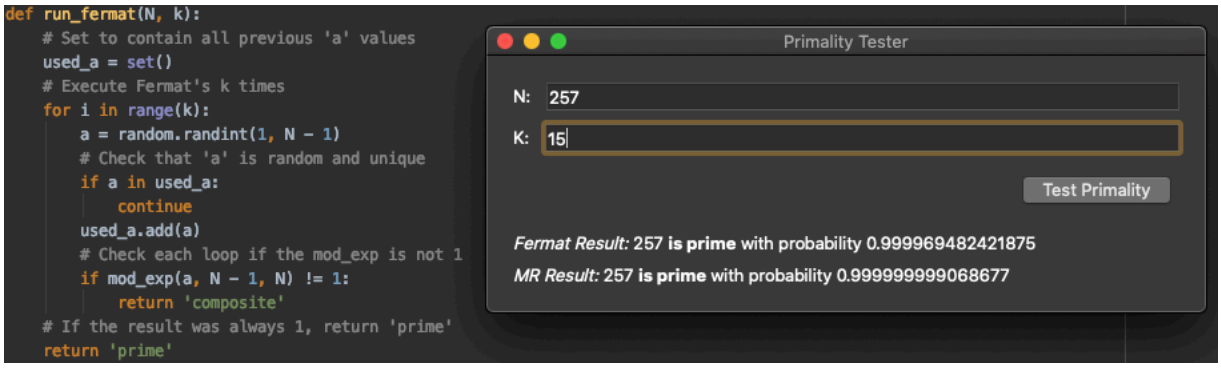
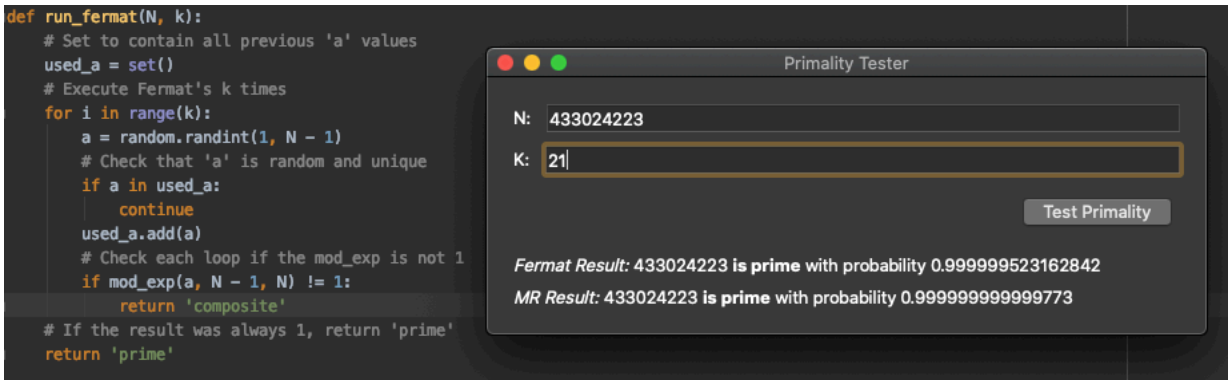
Jared Hanson
Project 1
CS 312 section 002

1) Screenshots of working application





2) Code with Comments to clarify solution
Time and Space complexity is all noted in the 3rd section of this document
Comments are added above each function explaining what its supposed to do.

```
import random


def prime_test(N, k):

    return run_fermat(N,k), run_miller_rabin(N,k)

#       Return x^y = ? (Mod N). It does this by splitting the function up over and over to
#       get a simpler function and then going back up the recursion.

def mod_exp(x, y, N):
    # Handle base case of 0
    if y == 0:
        return 1
    # Recursively call Mod EXP with half of y value
    z = mod_exp(x, (y//2), N)
    # Handle both even and odd Y cases
    if y % 2 == 0:
        return (z ** 2) % N
    else:
        # Odd case, account for the extra x
        return (x * (z ** 2)) % N

#       Return the probability of a wrong result (50%) compounded by k, the number of times
#       tested

def fprobability(k):
    # Return probability 1 - 1/2^k
    return 1 - (1 / (2 ** k))

#       Return the probability of a wrong result (25%) compounded by k, the number of times
#       tested

def mprobability(k):
    # Return probability 1 - 1/4^k
    return 1 - (1/(4 ** k))

#       Run Fermat's little theorem k times to get a more precise result. The theorem is run
#       using mod_exp and will return prime only if all the tests result as a 1.

def run_fermat(N, k):
    # Set to contain all previous 'a' values
    used_a = set()
    # Execute Fermat's k times
    for i in range(k):
        a = random.randint(1, N - 1)
        # Check that 'a' is random and unique
        if a in used_a:
            continue
        used_a.add(a)
        # Check each loop if the mod_exp is not 1
        if mod_exp(a, N - 1, N) != 1:
            return 'composite'
    # If the result was always 1, return 'prime'
    return 'prime'
```

```
#       Continuously do Fermat's theorem over and over dividing the exponent each time by
two. Stop doing this if you can no longer divide the exponent by two or the result is not 1.
Once stopped, check the result. If it is something other than 1 or -1, return composite. Run
this k times to up the precision, only return prime if all pass.

def run_miller_rabin(N, k):
    # Set to contain all previous 'a' values
    used_a = set()
    # Execute Miller-Rabin's k times
    for i in range(k):
        a = random.randint(1, N - 1)
        # Check that 'a' is random and unique
        if a in used_a:
            continue
        used_a.add(a)

        # Set variables to store current exponent and result of Mod-Exp
        result = 1
        exponent = N - 1

        #  Loop until result is something other than 1
        #  Loop starts with the first case of a ^ N - 1 and loops doing n ^ ½ each time.
        while result == 1:
            # Check that exponent is even
            if exponent % 2 == 0:
                result = mod_exp(a, exponent, N)
                exponent = exponent / 2

            # Current exponent is odd, triggers end of loop
            else:
                # Account for situation N - 1 is odd, but still enters the while loop
                if exponent == N - 1:
                    # If N - 1 is odd, the original number is even and thus not prime
                    return 'composite'
                break

        # Check if the result is not one of the two possible options 1 & -1
        if result != 1 and N - result != 1:
            return 'composite'
    # Return prime only if the algorithm never returns composite through k times.
    return 'prime'
```

3) Time and Space Complexity of solution (in bits)

Space

Time

Assuming that K is always much smaller than N (As mentioned by professor farrell)

Mod EXP(x, y, N): O(N^3) O(n)

Y//2 is a single right shift, meaning mod Exp will be called Y(bits) number of times or O(n) if n is the number of bits in y. The function that dominates are the equations attached to the return calls. Worst case contains 2 multiplications and a Modular function on N bits. These run at O(N^2) each. Run this N times and we get O(N^3)

The space is pretty much linear as data of size N is only stored in a variable z each time which will hold the computation. This will happen at each function call. O(n)

F_Probablility(K): O(k) (when compared to N, it's basically O(1)) O(1)

We're going to take a fraction and raise it to the k power. We can do this in O(log(k)) multiplications. If k has n bits, it's therefore O(n) multiplications. Since k is so much smaller than N is, it is relatively constant

Space is constant, no extra space is allotted O(1)

F_Probablility(K): O(k) (when compared to N, it's basically O(1)) O(1)

We're going to take a fraction and raise it to the k power. We can do this in O(log(k)) multiplications. If k has n bits, it's therefore O(n) multiplications. Since k is so much smaller than N is, it is relatively constant

Space is constant, no extra space is allotted O(1)

run_fermat(N, k): O(N^3) O(N)

This has a loop that will run K times and each time it will call modEXP O(N^3). Thus it runs at O(k*N^3). Searching the set for repetitions can be done in constant time O(1), so we can ignore that. Thus the whole thing runs at O(k*N^3), if K is much smaller than N, we can say it runs at O(N^3)

Space is O(N) because we have set to store each random value 'a' used once per loop. A will have some value of bits between 1 and N bits. So it would be O(K*N) or just O(N)

run_miller_rabin(N, k): O(N^4) O(N)

This function has a pair of nested loops. A loop that runs K times and a while loop that right shifts N-1 once each time, thus running N-1 (bits) times worst case. The while loop contains 1 Modular operation O(n^2), one ModEXP call O(N^3) and a constant time division O(1). Thus, the mod EXP dominates over the other functions
The rest of the function runs in constant time as it searches through a set for repeating variables and allots memory O(1).
Thus, we have a O(N^3) nested in two loops that run k times and N(bits) times. So it would be O(k*N^4) or just O(n^4)

Space is similar to fermats. We hold each random value 'a' in a set that will be size k. Each value will at most be N bits, so the space is O(k*N) or just O(N)

4) Explanation of equations used for probability functions:

Fermat's theorem has a success rate of 50% while Miller-Rabbin's is 75%. I first made sure that in my functions I was never re-using a test case (each 'a' in k tests was unique). Based on that fact, I could just use the equation 1/(2^k) and 1/(4^k) and then subtract that value from 1 for both functions respectively.