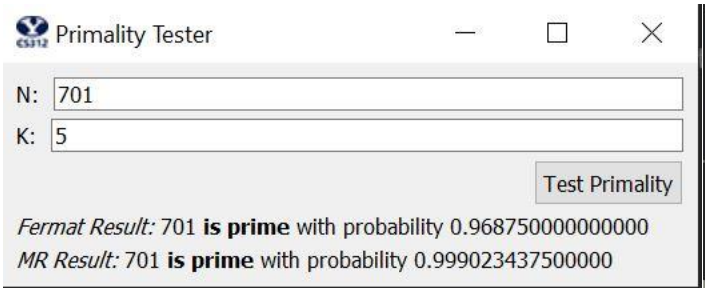

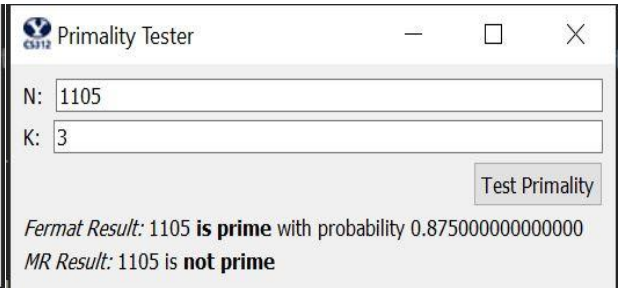


```python
def mod_exp(x, y, N):
    # You will need to implement this function and change the return value.

    if y == 0:
        return 1

    # Recursively call mod_exp function with y divided by 2 rounding down.
    z = mod_exp(x, (y//2), N)

    if y % 2 == 0:  # when y is even
        return (z**2) % N
    else:
        return (x*(z**2)) % N

def fprobability(k):
    # You will need to implement this function and change the return value.
    # Fermat's test returns YES when N is not prime: p(fail) <= (1/2^k), so
p(success) = 1 - p(fail)
    return 1 - 1/(2**k)

def mprobability(k):
    # You will need to implement this function and change the return value.
    # Miller-Rabin test passes when N is not prime : p(fail) <= (1/4^k), so
p(success) = 1 - p(fail)
    return 1 - 1/(4**k)

def run_fermat(N,k):
    # You will need to implement this function and change the return value, which
should be
    # either 'prime' or 'composite'.
    #
    # To generate random values for a, you will most likely want to use
    # random.randint(low,hi) which gives a random integer between low and
    #  hi, inclusive.

    # If N is 2, it is prime
    if N == 2:
        return 'prime'

    for i in range(1, k+1):
        # Create random number to test
```

```
        random_number = random.randint(1, N - 1)
        if mod_exp(random_number, N-1, N) != 1:
            return 'composite'
    return 'prime'




def run_miller_rabin(N,k):
    # You will need to implement this function and change the return value, which
should be
    # either 'prime' or 'composite'.
    #
    # To generate random values for a, you will most likely want to use
    # random.randint(low,hi) which gives a random integer between low and
    #  hi, inclusive.

    # Return 'prime' when N is 2
    if N == 2:
        return 'prime'

    # K tests
    for i in range(1, k+1):
        # Pick a random number 1 <= a < N
        random_number = random.randint(1, N-1)

        # Initial test
        if mod_exp(random_number, N - 1, N) != 1:
            return 'composite'

        # Initial test passed, compute the sequence and find the first value that
is not equal to 1
        y = (N - 1) / 2
        compute_sequence = True
        while compute_sequence:
            value = mod_exp(random_number, y, N)
            if value != 1:
                if (value + 1) != N:  # value is not -1
                    return 'composite'
                else:
                    break  # Passed the test, move on

            if y % 2 != 0:
                compute_sequence = False  # Can't square root anymore
            else:
                y = y / 2

    return 'prime'
```

3. For functions that calculate probabilities which are fprobability() and mprobability(), I would say that they have time complexity of O(1) which is constant time. Since the number of bits in k is very tiny compared to the number of bits n in N, we can say that they have time complexity of O(1) respect to n. Also, they just calculate the probabilities and return the resulting value, so both have space complexity of O(1).

For mod_exp(), the modular operation has time complexity of O(n^2), and multiplication and division also take O(n^2) time, and the function is calling itself n times recursively. Therefore, the time complexity of this function would be O(n^3). For space complexity, z is getting return value and it needs O(n) bits. Since the function recursively calls itself N times, the space complexity would be O(n^2).

For run_fermat() function, it calls mod_exp() function k times unless it breaks out of the loop in the middle. So, the time complexity would be O(kn^3). We store random number in the loop, but continuously call mod_exp(), so it would have the space complexity of O(n^2).

For run_miller_rabin() function, it calls mod_exp() in the for loop but use another loop inside to continue to square root of the number. Therefore, the time complexity for this function would be O(n^4). Since we keep calling mod_exp(), the space complexity would also be O(n^2).

4. fprobability() calculates the probability of success from running the run_fermat(). The probability that the testing primality function would say yes when N is not prime is less than or equal to (1/2)^k. Therefore, the probability that the algorithm would give the correct result would be 1 – (1/2)^k. Similarly, the probability that run_miller_rabin() would give wrong results is less than or equal to (1/4)^k. Therefore, p(success) would be 1 – (1/4)^k.