Primality Tester Write Up – Thomas Molina

The time complexity for Miller-Rabin is O(n^4) because you do k iterations of modular exponentiation, then another inner N-1 iteration of modular exponentiation, giving n^2 * n^2 in runtime. The space complexity is constant.

```python
def miller_rabin(N: int, k: int) -> str:
    """
    Time Complexity: O(n^4)
    Space Complexity: O(a + 2d + x + N) == O(1)
    """
    if N == 2:  # O(1)
        return 'prime'
    if N <= 1 or not N & 1:  # O(1)
        return 'composite'
    if N <= 3:  # O(1)
        return 'prime'

    d = N - 1
    while d % 2 == 0:  # O(1) because of constant 2
        d = d // 2
    d = int(d)

    iterations = k
    for i in range(iterations):  # O(k * (log(n) + n^2) * (N-1 * (n^2))) == k(n^2) * (N-1 * n^2) == O(n^4)
        a = random.randint(2, N - 2)  # log(n - 2)
        x = mod_exp(a, d, N)  # O(n^2)
        if x == 1 or x == N - 1:
            break
        while d != N - 1:
            x = mod_exp(x, 2, N)
            d *= 2
            if x == 1:
                return 'composite'
            if x == N - 1:
                break
    return 'prime'
```

For prime test, not including the space complexity for the functions that it contains, the total space complexity is constant, and the time complexity is O(n^4)

Modular exponentiation time complexity is O(n^2) and its space complexity is constant because it only needs 3 variables.

fprobability has an explanation attached to the function.

```python
import random


def prime_test(N: int, k: int) -> tuple:
    """
    time complexity: O(n^4) + O(n^2)
    space complexity O(1)
    """
    return fermat(N, k), miller_rabin(N, k)


def mod_exp(x: int, y: int, N: int) -> int:
    """
    Time complexity: O(n^2)
    Space Complexity: 3 variables (result, x, y) and 1 constant N, so O(x + y + N)
    but needs at most x*x space
    """
    result = 1
    x = x % N  # O(1) because N is constant
    while y > 0:
        if y & 1 == 1:
            result = (result * x) % N  # O(n^2)
        y = y >> 1  # O(1)
        x = (x * x) % N  # Time: O(n^2) because N is constant, Space: O(x ^ 2) b
    return result


def fprobability(k: int) -> int:
    """
    The probability of fermat's theorum picking a value that does not pass is 1/2 for each iteration.
    Because we go through k iterations, the probability that a value does not pass is 2^-k
    where k is the amount of numbers picked
    time complexity for the pow operator in python 3 is log k multiplications of n bit numbers, so it is n log(k)
     and subtraction of n bit numbers is O(n) time in total that is n + n log(k)
    space complexity: the function needs to store 1 + 2 * k bits to do multiplications of k
    """
    return 1 - pow(2, -k)
```

Fermat's theorem had a time complexity of O(n^2) because of modular exponentiation, and a space complexity of constant.

```python
def fermat(N: int, k: int) -> str:
    """
    Time Complexity: O(n^2) + k * (O(log n) + O(log n * a) + O(n^2) == O(n^2)
    Space Complexity: O(k + N + a)
    """
    if N == 2:  # O(1)
        return 'prime'
    # 0, 1, and even numbers are composite
    if not N & 1 or N <= 1:  # O(1)
        return 'composite'
    # 2, 3 are prime
    if N <= 3:  # O(1)
        return 'prime'
    if is_carmichael_number(N):  #  O(n^2)
        return 'carmichael'

    while k > 0:  # total = k * (O(log n) + O(log n * a) + O(n^2)
        a = random.randint(2, N - 2)  # O(log n)
        if gcd(N, a) != 1:  # time: O(n) bits, space:
            return 'composite'

        if mod_exp(a, N - 1, N) != 1:  # O(n^2)
            return 'composite'
        k -= 1
    return 'prime'
```

The rest of my code is either attached, or in these images:

```python
def is_carmichael_number(n: int):  # O(n^2)
    x = 2
    while x < n:
        if gcd(x, n) == 1:
            if pow(x, n-1, n) != 1:
                return False
        x = x + 1
    return True
```

For an n bit number (a) and m bit number (b) it would take O(log m * n) operations to complete the gcd, and would need O(m*n) space.

```python
def mprobability(k: int) -> float:
    """
    the upper bound for the miller-rabin test is 1/4^k
    time complexity for the pow operator in python 3 is log k multiplications of n bit numbers, so it is n log(k)
     and subtraction of n bit numbers is O(n) time in total that is n + n log(k)
    space complexity: the function needs to store 1 + 4 * k bits to do multiplications of k
    """
    return 1 - pow(4, -k)


def gcd(a: int, b: int) -> int:
    """
    Time Complexity: O(log a * b) in bits
    Space Complexity: needs O(a * b) for operations
    """
    if b == 0:
        raise ValueError("Cannot modulus by 0")
    if a < b:
        return gcd(b, a)
    elif a % b == 0:
        return b
    else:
        return gcd(b, a % b)
```