fermat.py

```python
import random
import math


def prime_test(N, k):
    # This is main function, that is connected to the Test button. You don't need to touch it.
    return fermat(N, k), miller_rabin(N, k)


# Time complexity: O(n^3). n recursive calls, each with multiplication of two n-bit numbers (n^2).
# Space complexity: O(n). no large variables, so that's constant. n recursions though, so space of O(n).
def mod_exp(x, y, N):
    # signals the end of the recursion
    if y == 0:
        return 1

    # recursion, halving y each time
    z = mod_exp(x, math.floor(y / 2), N)
    if y % 2 == 0:
        return z ** 2 % N
    else:
        return (x * z ** 2) % N


def fprobability(k):
    return 1 - (1 / 2) ** k


def mprobability(k):
    return 1 - (1 / 4) ** k


# Time complexity: O(k * n^3). using a for loop k times, calculating the mod_exp each time (n^3)
# Space complexity: O(k * n), using a for loop k times, doing mod_exp each time (space O(n)), so total is O(k * n)
def fermat(N, k):
    # use fermat's algorithm k times, return 'composite' if it ever fails, 'prime' if it does not
    for i in range(k):
        if mod_exp(random.randint(2, N - 1), N - 1, N) != 1:
            return 'composite'
    return 'prime'


# Time complexity: O(k * n^3 * log2(n)). mod_exp O(n^3), for loop O(k), recursion for the exp square root O(log2(n)).
# Space complexity: O(k * log2(n)). using for loop k times, recursion for the exp square root O(log2(n)).
def miller_rabin(N, k):
    # use miller rabin algorithm k times, return 'composite' if it ever fails, 'prime' if it does not
    for i in range(k):
        a = random.randint(2, N - 1)
        exp = N - 1

        # if it passes the fermat test, use miller rabin squaring of the exponent to continue checking.
        # if it continues to pass, continue the for loop. else return 'composite'
        if mod_exp(a, exp, N) == 1:
            passed_check = miller_rabin_helper(a, exp / 2, N)
            if passed_check:
                continue
        return 'composite'
    return 'prime'


# Time complexity: O(k * log2(n)). mod_exp O(n^3), worst case need log2(n) recursions for exp squaring.
# Space complexity: O(log2(n)). worst case need log2(n) recursions. no large variables, so that's constant.
def miller_rabin_helper(a, exp, N):
    if exp % 2 == 0:
        # calculate mod_exp for each recursion
        value = mod_exp(a, exp, N)
        if value == 1:  # continue recursion
            return miller_rabin_helper(a, exp / 2, N)
        elif value == N - 1:  # the value is equivalent to -1 (mod N), so end recursion. the number is probably prime.
            return True
        else:  # the value is not 1 or -1, so end recursion. the number is composite.
            return False
    else:  # exp is odd, so end recursion. the number is probably prime.
        return True
```