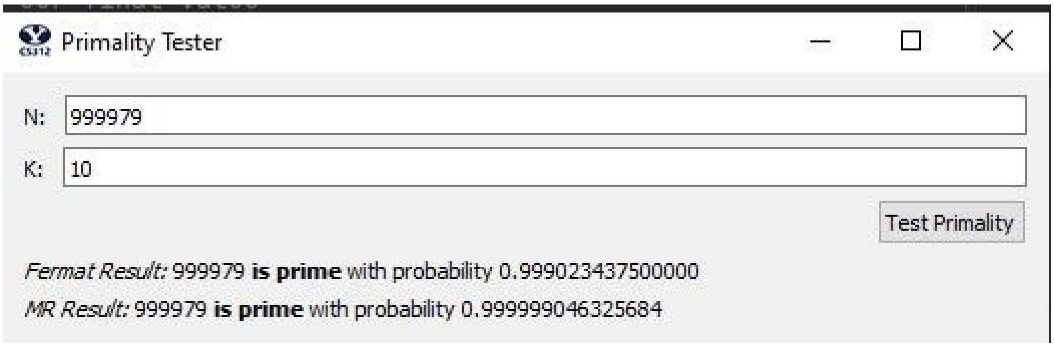
Alex Heiner- Lab 1 Submission

Part 1



Part 2

d) Carmichael numbers are good examples of inputs which can make the algorithms disagree. A Carmichael number can often trick the fermat test because if the fermat test chooses a base value which is relatively prime to the carmichael number, it will return composite. The Miller-Rabin test is a much more in depth test because you test a base value multiple times, and it checks if the return is 1 or N - 1. For small values of k, the fermat test often returns composite while the miller rabin test returns prime.

Part 3

Modular Exponentiation

<u>Function mod_exp(x,y,n)</u>

Input: Two n-bit integers x and N, and integer exponent y
Output: x^ymodN

If y == 0: return 1
z = modexp(x, floor(y/2), N)
if y is even: return z^2 mod N
else return x * z^2 modN

mod_exp explanation**:** The time complexity of this function is O(n^3). This is because there are n recursion calls of this function, and in each recursion call we are doing at least one multiplication which is O(n^2). The space complexity is O(n^2) because it is recursive so you have to store all of the O(n) multiplication space complexity.

Fermat Algorithm
Function fermat(k,N)
Input: Positive integers k and N
Output: yes/no

Loop while k > 0
Choose random positive integer 0 < a < N
If a^(N-1) != 1 modN: return no
k = k - 1

Fermat explanation: The time complexity of the fermat function is O(kn^3). This is because we are completing k number of tests, and in each test we are doing modular exponentiation one time in each test which is O(n^3). The space complexity is O(n^2) because you are calling mod_exp which uses space complexity of O(n^2).

Miller Rabin Algorithm
Function miller_rabin(k,N)
Input: Positive integers k and N
Output: yes/no

Choose random positive integer 0 < a < N
If a^(N-1) != 1 modN: return no
Else: set boolean value is_prime = miller_rabin_helper(a, N - 1, False)
If is_prime is true: continue in loop and decrement k value
Else: return no

Function miller_rabin_helper(a, exp, N, encounter)
Input: positive integers a, exp, and N. Boolean encounter set to false
Output: yes/no

If exp is odd: return True

If a^(N-1) = N-1: encounter is set to true

If a^(N-1) DNE 1 and a^(N-1) DNE N - 1:
    If encounter is true: return no
    Else: return yes
Else: return miller_rabin_helper(a, exp/2, N, encounter)

Miller Rabin explanation: The time complexity of these two functions is O(kn^4). The miller_rabin function performs k number of tests, which means that it calls the miller_rabin_helper function k times. The worst case time complexity of the miller_rabin_helper function is O(n^4) because we call the mod_exp function (which is of order O(n^3)), n times in a worst case scenario. The time complexity of the miller_rabin_helper function is just O(n^3). Making the total time complexity of the miller rabin algorithm O(kn^4). The space complexity is O(n^2) because you are calling mod_exp which uses space complexity of O(n^2).

Probability Functions

Function fprobability(k)
Input: positive integer k
output : Fermat probability

Return 1 - (1/2^k)

fprobability explanation: Time complexity of O(kn^2) because we are multiplying 1/2, which is order O(n^2), k times.

Function mprobability(k)
Input: positive integer k
output : Miller Rabin probability

Return 1 - (1/4^k)

mprobability explanation: Time complexity of O(kn^2) because we are multiplying 1/4, which is order O(n^2), k times.

Part 4

Fermat Probability:

The formula for calculating the probability that the fermat algorithm is correct is written as: 1 - 1/2^k, where k represents the number of tests performed. If you are testing a number N, a^N-1 = 1mod N for at most half of the values between a and N. This means that you have a 50% chance of revealing a composite N if you choose one value for a. If you perform k = 2 tests your probability of being correct is 1 - (1/2 * 1/2). This would be true for any value of k > 0 that you choose.

Miller Rabin Probability:

The Miller Rabin test combines the Fermat algorithm with a square root test on the exponent to give us a more in depth test. In this case, if you are testing a number N, 3/4 of the values between 1 and N - 1 will be able to reveal a composite N. This means that we have a 75% chance of revealing a composite if we undergo one test of N. If we increase the number of tests to 2, our probability of revealing a composite N is 1 - (1/4 * 1/4). The probability formula can be modeled as 1 - 3/4^k for k > 0

Appendix

```
import random


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 of n^3. N recursion calls and each recursion you are multiplying which is
# order O(n^2). The space complexity is O(n^2) because it is recursive so you have to store
# all of the O(n) multiplication space complexity.

def mod_exp(x, y, N):
   # When we are finished with recursion
   if y == 0:
      return 1
   z = mod_exp(x, y//2, N)

   # Begin returning values from the bottom up until we get our final value
   if y % 2 == 0:
      return z ** 2 % N
   else:
      return x * z ** 2 % N


# Time complexity of kn^2 because we are multiplying n bit number k times. Space
# complexity is O(n) because we are multiplying 2 n bit numbers which results in an n bit
# number

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


# Time complexity of kn^2 because we are multiplying n bit number k times. Space
# complexity is O(n) because we are multiplying 2 n bit numbers which results in an n bit
# number

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

```
# Time complexity of kn^3 because we call mod_exp (which is order n^3) k times. Space
# complexity is O(n^2) because we the recursion needs to store each multiplication
operation # which is order O(n)

def fermat(N, k):

    # if N is an even number return composite
    if N % 2 == 0:
        return 'composite'

    # Run through the number of tests
    while k > 0:

        # select a random base and compute modular exponentiation
        rand_num = random.randint(1, N - 1)
        num = mod_exp(rand_num, N - 1, N)

        # If the result of the modular exponentiation is not 1- we know the number is composite
        if num != 1:
            return 'composite'
        k -= 1

    # Return prime if we didn't return composite
    return 'prime'


# Time complexity of kn^3 because we call mod_exp (which is order n^3) k times. The
# space complexity is O(n^2) because you are calling mod_exp which uses space
# complexity of O(n^2).
def fermat(N, k):

    # if N is an even number return composite
    if N % 2 == 0:
        return 'composite'

    # Run through the number of tests
    while k > 0:

        # select a random base and compute modular exponentiation
        rand_num = random.randint(1, N - 1)
        num = mod_exp(rand_num, N - 1, N)

        # If the result of the modular exponentiation is not 1- we know the number is composite
        if num != 1:
            return 'composite'
        k -= 1
```

```
    # Return prime if we didn't return composite
    return 'prime'


# Time complexity of kn^4 because we call miller_rabin_helper() k times and it is order
#  O(n^4). The space complexity is O(n^2) because you are calling miller_rabin_helper()
# which uses space complexity of O(n^2).

def miller_rabin(N, k):

    # If number is even we know it is composite
    if N % 2 == 0:
        return 'composite'

    # Set exponent equal to test value minus one
    exponent = N - 1

    while k > 0:

        # Select random base, compute modular exponentiation to see if it passes the first test
        rand_num = random.randint(1, N - 1)
        num = mod_exp(rand_num, exponent, N)
        if num == 1:

            # if N passes the first test call helper function to run miller rabin test until it returns
            encounter = False
            is_prime = miller_rabin_helper(rand_num, exponent, N, encounter)
            if not is_prime:
                return 'composite'
        else:
            return 'composite'
        k -= 1

    return 'prime'


# Time complexity of O(n^4) because we are doing n recursion calls and calling mod_exp()
# each time which is order O(n^3). The space complexity is O(n^2) because you are calling
# mod_exp() which uses space complexity of O(n^2).


def miller_rabin_helper(rand_num, exponent, test_num, encounter):

    # If the exponent is odd we want to move on to test the next base
    if exponent % 2 != 0:
        return True

    result = mod_exp(rand_num, exponent, test_num)
```

```
if result == test_num - 1:
    encounter = True

# if the result of mod_exp is not one and it is also not test_num - 1 return false if we have
# not seen a test result be test_num - 1

if result != 1 and result != test_num - 1:
    if not encounter:
        return False
    else:
        return True

else:
    return miller_rabin_helper(rand_num, exponent/2, test_num, encounter)
```