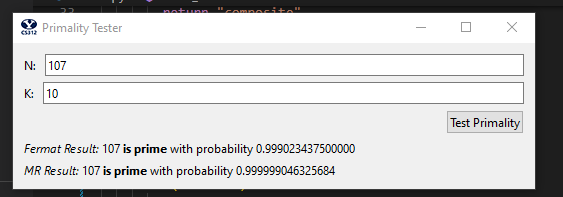
**Primality Testing – Lab Submission**

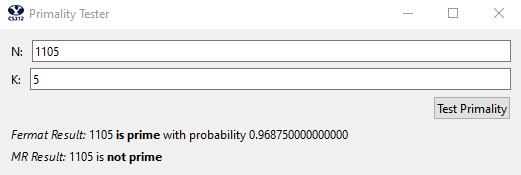
Part 1.



Part 2. See appendix below

Part 3.

What I did to test where Fermat’s and Miller-Rabin’s algorithms gave different answers was I started to test the Carmichael numbers. This is because the Carmichael numbers are by definition a composite number that fails the Fermat prime test for all values “a” that are relatively prime to it. I tried a Carmichael number (1105) with different values of k. I noticed that as k got bigger (10 or so) then both Fermat’s and Milller-Rabin’s algorithms correctly caught this number as not prime. But when the k was lower, like 5, then usually the Fermat’s algorithm got fooled but the Miller-Rabin algorithm caught the number correctly. This is the case because Fermat fails when the “a” value chosen is relatively prime to the N value, like said above. Choosing just a few “a” values increases the likely hood of it being relatively prime, but as soon as it is not relatively prime, then Fermat catches it. The bigger the k, the more likely one of these will not be relatively prime. On the other hand, Miller-Rabin makes more tests with this one “a” thus having more of a chance to get it correct.



Part 4.

Modular Exponentiation

* Time: To get the time complexity, we have to first see how long each recursive step takes. The y % 2 part is order n^2 because of mod. z\*z is another n^2 operation with % N making one more n^2. All of this together is 3n^2. This is the amount of time for one recursive call, but adding up all of the recursive calls which happens for each bit of y, we multiply this by n. This gives the total time complexity to 3n^3 or O(n^3).
* Space: Since we are storing the variable x and y recursively n times (z is stored, but it does not need to be put on the stack), we will have O(n^2) complexity

Fermat’s Algorithm

* Time: The algorithm repeats itself inside of a for loop k times. So first we need to figure out just one pass of the algorithm. Doing the N-1 subtraction is O(n) and taking the modular exponentiation like we saw above is O(n^3). This just simplifies to an order of n^3. Do this k times and we have O(kn^3)
* Space: We need two variables “a” and “mod” and these variables get reused in the for loop. This means that our space complexity is the complexity of mod\_exp or O(n^2).

Miller-Rabin’s Algorithm

* Time: Like Fermat’s algorithm, we are doing Miller-Rabin k times inside of a for loop so we have to compute the time for one loop. First we see N – 1 twice which is 2n time. Then we have to add on the time for the miller-rabin-helper function. This function does modular exponentiation which is n^3. Then it does e % 2 which is n^2 giving a total of n^3 + n^2 time. Doing this n times recursively (because we are shifting the bit until we hit an odd number) we have a total of n^4 + n^3. Adding this to what we already have, this gives us n^4 + n^3 + 2n or O(n^4). Doing this k times in the loop gives us O(kn^4).
* Space: We are storing 4 variables throughout this process: a, N, e, and mod. Since the mod\_exp function requires O(n^2), we will need overall O(n^2) complexity of space

Fermat Probability

* Time: This is 1 subtraction (O(n)), k multiplications (O(k\*n^2)) and one division (O(n^2)) giving a total of O(n^2).
* Space: Since k is the only variable we are storing, it is O(k)

Miller-Rabin Probability

* Time: This is the same as the Fermat Probability, just a different constant factor so O(n^2)
* Space: Same as before: O(k)

Part 5.

The Equation I got for the probability of Fermat’s algorithm was 1-.5^k. This equation came from the fact that there is a .5 chance that any chosen “a” value will be prime. Then the next time we pick an “a” value, it will be another .5 chance of our .5 chance. This perpetuates k times or .5^k. But this is the probability that the Fermat algorithm gave us the wrong answer. To get the probability that it gave us the right answer, we just do the inverse (1-.5^k). For the Miller-Rabin algorithm, we know that there is a .75 chance that for composites it gives us correct results. This means that there is a .25 chance that a composite number was incorrectly identified. Each subsequent “a” value chosen is .25 chance, or .25^k. Inverting this to get the probability that we did it correctly is 1-.25^k.

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: Has n recursive calls and does 2 or 3 multiplies and one division at each call, so overall

# it has complexity O(n^3)

# Space: Since we have to store x and y recursively n times, we will have O(n^2) space

def mod\_exp(x, y, N):

  if y == 0:

    return 1

  z = mod\_exp(x, y // 2, N)

  even = y % 2

  if even == 0:

    return z \* z % N

  else:

    return z \* z \* x % N

#Time: subtraction (n), multiplications (k\*n^2) and division (n^2) is O(k\*n^2)

#Space: We only have to store k, so O(k)

def fprobability(k):

    return 1 - .5\*\*k

#Time: subtraction (n), multiplications (k\*n^2) and division (n^2) is O(k\*n^2)

#Space: We only have to store k, so O(k)

def mprobability(k):

    return 1 - .25\*\*k

#Time: The dominant factor in each loop is the mod\_exp which is O(n^3).

# Doing this k times is O(k\*n^3)

#Space: We do not need to store anything on the stack, so we will have the space of our biggest

# variable which is mod\_exp or O(n^2)

def fermat(N,k):

    for i in range(k):

      a = random.randint(2, N - 1)

      mod = mod\_exp(a, N-1, N)

      if mod != 1:

        return "composite"

    return "prime"

#Time: Our complexity is the complexity of the help function O(n^4) k times or O(k\*n^4)

#Space: since we are reusing variables, our complexity is that of the helper function or O(n^2)

def miller\_rabin(N,k):

  for i in range(k):

    a = random.randint(2, N - 1)

    if miller\_rabin\_helper(a, N - 1, N) == "composite":

      return "composite"

  return "prime"

#Time: We have n recursive calls of a dominant mod\_exp O(n^3). This makes O(n^4) in all

#Space: Since we are resuing the variables and none are on the stack, we just have the complexity

# of mod\_exp which is O(n^2)

def miller\_rabin\_helper(a, e, N):

  mod = mod\_exp(a, e, N)

  if (mod != 1):

    #mod == -1 and mod = N - 1 are the same

    if mod != N - 1:

      return "composite"

    else:

      return "prime"

  #If we cannot divide by two anymore then we are done

  if e % 2 != 0:

    return "prime"

  return miller\_rabin\_helper(a, e // 2, N)