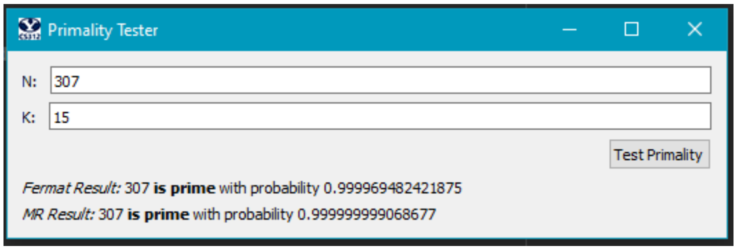
Trevor Ashby

Fermat's Primality Test

1. Working Example

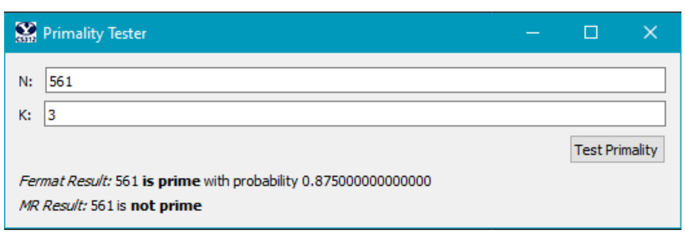


2. Code that I wrote

a.b.c. See Appendix

d. Some experimentation I did to identify inputs where the two algorithms disagreed was using the number 561 (which is a Carmichael number) and the value K = 3. At first I tested with the 561 Carmichael number and K = 10. These numbers did not cause any disagreement in the algorithms. I figured that using a Carmichael number was a good start because that is the weakness in the Fermat algorithm. So I decided to begin playing around with the K value until I found that 3 was a good number for seeing the algorithms disagree. I think this is because when we use K = 10 the accuracy increases due to using a greater pool of random numbers that must pass all tests. Whereas with only 3 random numbers being tested, it is much easier to slip through the a^exp mod N = 1 filter based on specific random numbers being chosen. Below is a screenshot of said disagreement.

The reason that Miller-Rabin is more accurate than Fermat's, is if the random value A is relatively prime to N, it will then cause it to "pass" Fermat's. However, Miller-Rabin checks for non-trivial roots which can only be found in composite numbers. This means that even if A is relatively prime to N, it will still be detected as a composite number.



3. Time and Space Complexity

<table>
<tr><td>

Pseudo-code
Mod_exp(x, y, N):
    If y = 0. Return 1
    Z = mod_exp(x, y/2, N)
    If (y is even) return z^2 mod N
    Else return x * z^2 mod N

</td><td>

Space complexity: O(N^2)
For each call of mod_exp, N space is required. Because mod_exp is recursive for N times, the space complexity is O(N^2).
Time Complexity: O(N^3)

</td></tr>
</table>

	Because mod_exp is doing division for each call, this means that the time complexity is O(N^3).
<u>Pseudo-code</u> Fermat(N, K): Generate k number of random values For each random value If mod_exp(randomval, N – 1, N) != 1 return ‘comp’ If none are ‘comp’ return ‘prime’	**<u>Space Complexity: O(N^2)</u>** For the random number generation, k space is required, but can be ignored because it is a constant. The for loop which causes mod_exp to run will require k * N^2 space due to the mod_exp. (k to be ignored). Therefore space complexity is O(N^2). **<u>Time Complexity: O(N^3)</u>** For the random number generation, it will take k time to execute, but can be ignored because it is a constant. The for loop which calls mod_exp will run in k * N^3 time. Again, ignoring the k constant. Therefore time complexity is O(N^3).
<u>Pseudo-code</u> Miller_rabin(N, K): Generate k number of random values For each random value X = mod_exp(randomval, N – 1, N) If x != 1 return ‘comp’ Exp = N – 1 While Exp is even Exp /= 2 X = mod_exp(x, exp, N) If x != -1 and x != 1 return ‘comp’ If none are ‘comp’ return ‘prime’	**<u>Space Complexity: O(N^3)</u>** For the random number generation, k space is required, but can be ignored because it is a constant. The for loop which contains the rest of the code will take k space, which is ignored because it is a constant. The first mod_exp will take N^2 space. Next the while loop which calls the second mod_exp will take N space for the loop and N^2 space for the mod_exp. Therefore, together they will take N^3 space. Due to the max rule the space complexity is then O(N^3). **<u>Time Complexity: O(N^4)</u>** For the random number generation, it will take k time to execute, but can be ignored because it is a constant. The for loop which contains the rest of the code will take k time, but that too can be ignored because it is a constant. The first mod_exp will take N^3 time to execute. Next the while loop which calls the second mod_exp will take N time to execute and N^3 time for the mod_exp within to execute. Therefore, together they will take N^4 time to execute. Due to the max rule the time complexity is then O(N^4).

4. Discuss Probabilities of Correctness	
<u>Fermat:</u> *Equation: 1 – [1 / (2^k)]* I used this equation because Fermat’s algorithm has at least a ½ chance for correctness. This means that for the greater number of ‘k’ that is used, the more probable that the algorithm is correct. Thus approaching 100% with greater values of ‘k’.	**<u>Miller-Rabin:</u>** *Equation: 1 – [1 / (4^k)]* I used this equation because Miller-Rabin’s algorithm has at least a 3/4 chance for correctness. This means that for the greater number of ‘k’ that is used, the more probable that the algorithm is correct. Thus approaching 100% with greater values of ‘k’.

Appendix

```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)
# SPACE COMPLEXITY: O(n^2)
def mod_exp(x, y, N):
    # If the exponent is 0, then 1 can be returned and no calculations necessary
    if y == 0:
        return 1
    # Recursively call mod_exp for x and half of
    # the exponent y.
    z = mod_exp(x, math.floor(y/2), N)
    # If y is even, then return z squared mod N
    if (y % 2) == 0:
        return math.pow(z, 2) % N
    # If y is odd, return x times z squared mod N
    else:
        return (x * (math.pow(z, 2))) % N

# TIME COMPLEXITY: O(1)
# SPACE COMPLEXITY: O(1)
def fprobability(k):
    return 1.00 - (1 / (math.pow(2, k)))

# TIME COMPLEXITY: O(1)
# SPACE COMPLEXITY: O(1)
def mprobability(k):
    return 1.00 - (1 / (math.pow(4, k)))

# TIME COMPLEXITY: O(n^3)
# SPACE COMPLEXITY: O(n^2)
def fermat(N,k):
    randvals = []
    # For k amount of times, generate a random integer from 2 -> N-1
    # and put it into the randvals list. || TIME COMPLEXITY: O(k)
    for i in range(k):
        randvals.insert(i, random.randint(2, N - 1))
    # For each of the randvals, calculate the mod_exp of it with N-1. If this does not equal 1,
    # then the number is composite. || TIME COMPLEXITY: O(k)
    for a in randvals:
        if (mod_exp(a, N - 1, N)) != 1: # TIME COMPLEXITY: O(n^3)
            return 'composite'
    # If the above loop never returns, this means that all the
    # random numbers passed and that the number is prime.
    return 'prime'

# TIME COMPLEXITY: O(n^4)
# SPACE COMPLEXITY: O(n^3)
def miller_rabin(N,k):
    randvals = []
    # For k amount of times, generate a random integer from 2 -> N-1 and put it
    # into the randvals list. || TIME COMPLEXITY: O(k)
    for i in range(k):
        randvals.insert(i, random.randint(2, N - 1))

    for a in randvals:
        # For each of the randvals, calculate the mod_exp of it with N-1. If this does not equal 1,
        # then the number is composite. || TIME COMPLEXITY: O(k)
```

```
        x = mod_exp(a, N - 1, N) # TIME COMPLEXITY: O(n^3)
        # If the mod_exp of the randval and N-1 is not 1, return composite.
        if x != 1:
            return 'composite'
        # This exp value is for tracking the changing exponent which starts as N-1
        exp = N - 1
        # Continue this process while exp is even, and each loop
        # divide exp by 2.
        while (exp % 2) == 0: # TIME COMPLEXITY: O(n)
            exp /= 2
            # Calculate x to the exp power mod N. If this does not equal -1 or 1,
            # then the number is composite. || TIME COMPLEXITY: O(n^3) <----
            x = mod_exp(x, exp, N)
            if x != -1 and x != 1:
                return 'composite'
    # If the above loop doesn't trigger any returns, then the number must be prime.
    return 'prime'
```