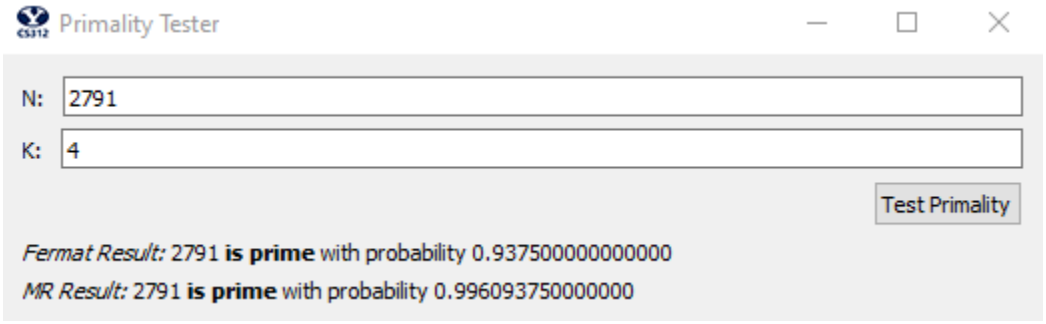
Matthew R. Christensen

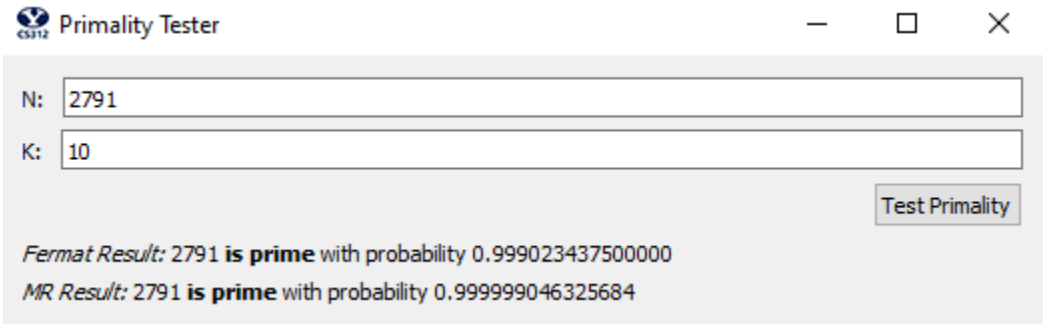
CS 312 (001) – Dr Martinez, Tony R

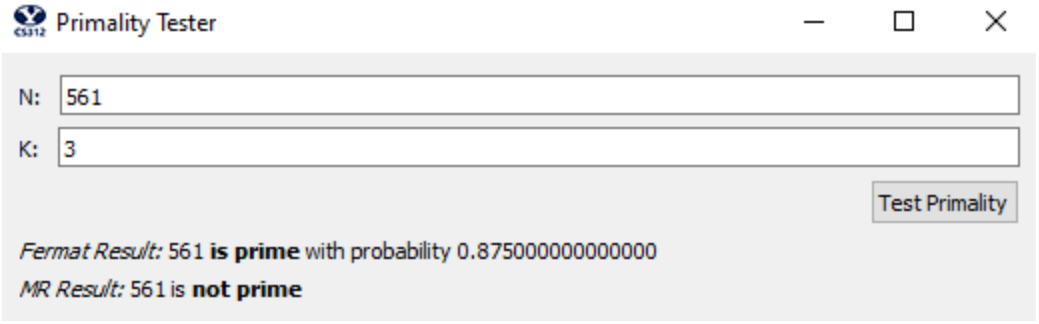
January 25, 2021

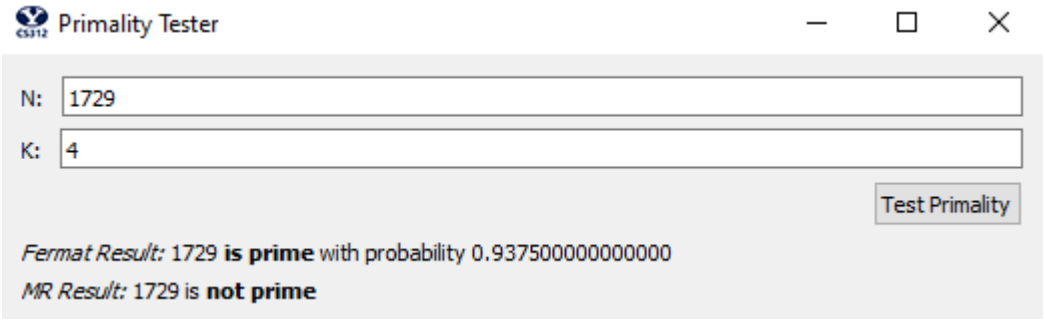
Fermat (and Miller-Rabin) Algorithm Report

Screenshots









Complexity

If the level of detail is not granular enough in this section (from time and space complexity) please reference the code in the appendix for more detail and specifics with big O.

Firstly, we have the complexity for *mod_exp*:

//Runs y times, let it be y be equivalent to n

mod_exp(x, y, N):

if y == 0: return 1

z = mod_exp(x, y/2, N) //Time Complexity: division = O(n^2)

*if y % 2 == 0: return (z ** 2) % N //Time Complexity: power = O(z(n^20) = O(n^2)*

*else: return (x * (z ** 2)) % N Time Complexity: mult and exp = O(n^2 + z(n^2)) = O(n^2)*

For time complexity, we get O(n(n^2+z(n^2))), because of the function running n times, and withing each call, with the heaviest operations have a n^2 multiplication and a z(n^2) power. We can drop the constant and simplify it to = O(n^3)

The space complexity will be similar. We run the algorithm n times and have 3n space for the stored variable (such as the arguments and z, think stack size). This simplifies O(n(3n)) = O(n^2)

Next are the complexities for the *probability* functions. Because they are both the same, we will use a more general abstracted function (where *RATIO* is the desired ratio of correct probability) to talk about both the Fermat and Miller-Rabin probability functions at the same time.

probability(k):

return 1 – RATIO^k

For time complexity, we have a division and power multiplication, which is O(n^2 + (k)n^2), which is simplified to = O(n^2)

Space complexity is simple constant, or O(1)

Next is the *Fermat* function.

fermat(N, k):

for all a_i *|* $i = 1, 2, \ldots, k$ *and* $a_i < N$*:*

if mod_exp(a_i*, N-1, N) != 1: return composite*

return prime

Firstly, for time complexity, we know that the look will run k times, and each time it calls *mod_exp* (which we know is O(n^3) from our previous calculations. Thus, Fermat will be O(k(n^3)), which simplifies to simply O(n^3) because we drop the constant factor.

For space complexity, we do the exact same thing. The loop runs k times, with *mod_exp* being called each time with a space complexity of O(n^2). Thus, for Fermat, the space complexity is O(k(n^2)) = O(n^2).

Finally, we calculate the time and space complexity for the *Miller-Rabin* function.

miller_rabin(N, k):

for all a_i *|* $i = 1, 2, \ldots, k$ *and* $a_i < N$*:*

if mod_exp(a_i*, N-1, N) != 1: return composite //equiv to fermat: Time O(n^3), Space: O(n^2)*

exp = N-1

while exp is even: // runs log(exp) times (for time and space complexity)

squares.add(mod_exp(a, exp // 2, N)) //Time Complexity: n^3, Space Complexity: n^2

exp = exp/2 //Division is O(n^2)

for all sqrt in squares:

if sqrt != 1:

if sqrt == N – 1: break

else: return composite

return prime

For time complexity, we don't need to worry about the for loop of squares, because the biggest operation will be trumped by operations in the exp loop. Knowing that *mod_exp* is O(n^3) for time complexity, we find the time complexity, to be O(k(n^3 + log(exp)((n^3 + n^2)))) = O(n^3 + log(exp)(n^3)) = O(log(exp)(n^3)). This is because to constant factors go away, and we take the dominating factors.

And for space complexity, it's pretty much the same process but with *mod_exp()* having a space complexity of O(n^2). This means big O is, O(k(n^2 + log(exp)(n^2 + n^2))) = O(n^2 + log(exp)(n^2)) = O(log(exp)(n^2)).

Again, if more granularity is desired, please reference the appendix as the code has much more detail with the space and time complexity.

Discussion

Firstly, we will discuss when the two algorithms (Fermat and Miller-Rabin) disagree whether a particular number is prime or composite.

The Fermat test is pretty good at finding prime numbers, except for when it comes to Carmichael numbers. Carmichael numbers are composite (non-prime) numbers that often fool the Fermat algorithm to think they are prime. This is because a^(N-1) ≡ 1 (mod N) | N is a Carmichael number with all values of a that

are relatively prime to N (again, where N is a Carmichael number). Carmichael numbers have also been proven to be infinite.

The Rabin-Miller algorithm/test is a more refined way of finding primes, and does so in such a way to avoid miscataloging Carmichael numbers are prime. If the number passes the initial Fermat test, then a follow up test is executed: by looking at the sequence of square we can see were we ran into the first value that wasn't 1. If that value is N-1 than it may still be prime, and we continue testing. However, if it is not equivalent to N-1 we know that the number must be composite, there by finding Carmichael numbers to be composite (instead of tricking the Fermat system into think they are prime). This does however take more resources in time and space complexity (as described above in the Complexity section).

This can be seen in screenshots four and five. 561 is the smallest Carmichael number, with its factors of: 3*11*17=561 (which proves it is not a prime number). However, our Fermat algorithm claims it is with a probability of .875 (with k=3) while our implementation of the Miller-Rabin algorithm correctly identifying the number as composite. This is the same case with screenshot five. The composite, Carmichael number, of 1729 is incorrectly identified as prime with the Fermat algorithm, but correctly categorized as composite by the Miller-Rabin algorithm.

It is important to realize, however, that the Fermat test will not always incorrectly categorize Carmichael numbers. Many times, especially with a k value of 3 or higher, the Fermat algorithm will correctly identify Carmichael numbers as composite.

Lastly, we will now describe the two equations we used to compute the probability p, where p is the correctness of finding a certain number to be prime with either the Fermat or Miller-Rabin algorithm.

For the Fermat probability of correctly identifying a prime, when know that all prime numbers will pass the Fermat test. However, we do have some false positives when the algorithm incorrectly identifies composite numbers as primes. Because of the one-to-one nature of the problems, we imply that N fails the Fermat test for at least half of the values of a. We can get minimize this one sided error if we repeat the algorithm for many values of a if they are unique and randomly selected, which brings down to probability to 1-1/2^k.

With the Miller-Rabin algorithm, we can improve our accuracy with the square-root check, which an improvement of up to 3/4 probability for the random values of a that are greater than 1 and less than N-1, and this even takes into account Carmichael numbers. This leaves us a probability of 1-1/4^k, which is a significant improvement over the probability of the Fermat test.

Thank you.

Appendix

fermat.py:

```
import random



def prime_test(N, k):
    return fermat(N, k), miller_rabin(N, k)
```

```
# Time Complexity: O(n(n^2+n^2)) = O(n^3)
# Space Complexity: O(n(3n)) = O(n^2)
def mod_exp(x, y, N):  # Algorithm from Figure 1.4
    if y == 0:  # Time Complexity: Runs y times, let it be n
        return 1

    z = mod_exp(x, y // 2, N)  # Time Complexity: division = O(n^2)
    if y % 2 == 0:  # If y is even
        return (z ** 2) % N  # Time Complexity: power = O(z(n^20) = O(n^2)
    else:  # If y is odd
        return (x * (z ** 2)) % N  # Time Complexity: mult and exp = O(n^2 + z(n^2)) = O(n^2)


# Time Complexity: Divide and power = O(n^2 + (k)n^2) = O(n^2)
# Space Complexity: O(1)
def fprobability(k):  # From pg 28
    return 1 - (1 / 2) ** k


# Time Complexity: Divide and power = O(n^2 + (k)n^2) = O(n^2)
# Space Complexity: O(1)
def mprobability(k):  # From pg 28
    return 1 - (1 / 4) ** k


# Time Complexity: O(k(n^3)) = O(n^3)
# Space Complexity: O(k(n^2)) = O(n^2)
def fermat(N, k):  # Algorithm from Figure 1.8 (pg 27)
    if N == 1:
        return "composite"
    if N == 2:
```

```
        return "prime"


    # Time Complexity: runs k times
    for a in random.sample(range(2, N), min(k, N - 2)):  # Iter through a1, a2, ... , ak | 1 < a < n, and a is unique and random
        if mod_exp(a, N - 1, N) != 1:  # Time Complexity: O(n^3), Space Complexity: O(n^2)
            return "composite"


    return 'prime'


# Time Complexity: O(k(n^3 + log(exp)(n^3 + n^2))) = O(n^3 + log(exp)(n^3)) = O(log(exp)n^3)
# Space Complexity: O(k(n^2 + log(exp)n^2)) = O(log(exp)n^2)
def miller_rabin(N, k):  # From pg 28
    if N == 1:
        return "composite"
    if N == 2:
        return "prime"


    # Time and Space Complexity: runs k times
    for a in random.sample(range(2, N), min(k, N - 2)):  # Iter through a1, a2, ... , ak | 1 < a < n, and a is unique and random


        # Equivalent to the Fermat Test
        res = mod_exp(a, N - 1, N)  # Time Complexity: O(n^3)
        if res != 1:
            return "composite"


        listofSqrt = []
        exp = (N - 1)


        # Time and Space Complexity: runs log(exp) times
        while (exp % 2 == 0):  # Square root until you get to an odd exponent
            listofSqrt.append(mod_exp(a, exp // 2, N))  # Space Complexity: n^2, Time Complexity: n^3
            exp = exp // 2  # Time Complexity: division = O(n^2)
```

```
        # Time Complexity: will be neglectable, because at most we just do a subtraction
        for sqrt in listofSqrt:  # Go through list of squares
            if sqrt != 1:  # If it's not 1
                if sqrt == N - 1:  # subtraction = O(1)
                    break  # May be prime, lets keep going
                else:  # And if it's not N-1 (or -1 % N - they are equivalent)
                    return "composite"  # Than it's composite

    return 'prime'  # We've gone through the a's and it look prime

# test
# for i in range(1,100):
#     if miller_rabin(i, 3) == "prime":
#         print(i)
```