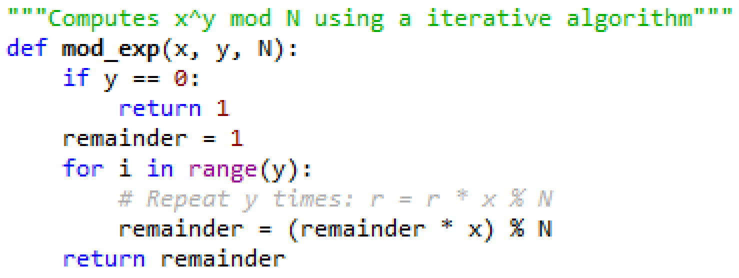
Exploration of Fermat and Miller-Rabin Primality Algorithms

Katherine Wilsdon

Idaho State University

Modular Exponentiation

Originally, a recursive modular exponentiation algorithm was written. But, the Carmichael number of 1729 gave a runtime warning that an overflow was encountered in int_scalars. The recursive function called itself so many times that the space to store the variables exceed the amount provided by the stack. The implementation was switched to an iterative algorithm where a variable equal to 1 is multiplied by *a* mod *N*. The variable equals the result of the modulation and then the algorithm repeats for *y* times.

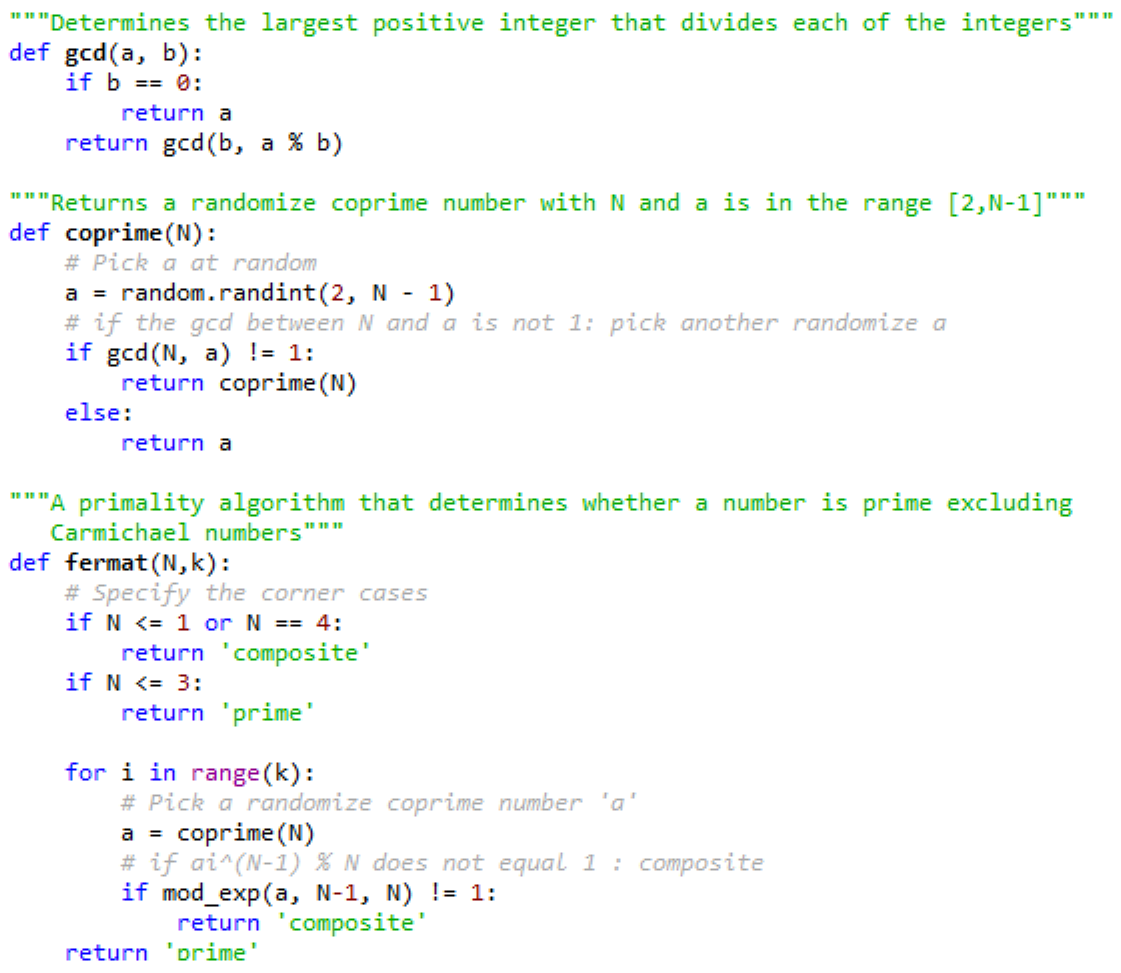


```
"""Computes x^y mod N using a iterative algorithm"""
def mod_exp(x, y, N):
    if y == 0:
        return 1
    remainder = 1
    for i in range(y):
        # Repeat y times: r = r * x % N
        remainder = (remainder * x) % N
    return remainder
```

Figure 1: Modular Exponentiation Algorithm.

Fermat Primality Algorithm

The Fermat primality algorithm states that if p is prime, then $a^{N-1} \equiv 1 \bmod N$ for any a such that $1 \leq a < N$. For the algorithm provided, the range of *a* was $2 \leq a < N$. 1 was not used because 1 to the power of some number is always 1. In a Carmichael-free universe, $a^{N-1} \not\equiv 1 \bmod N$ for some *a* relatively prime to *N*, then it must hold for at least half the choices of $a < N$. Because of this scenario, a randomized coprime number, *a*, was determined between *a* and *N* where the $gcd(a, N) = 1$. A coprime number, a, was used to eliminate the chances of randomly getting a Fermat witness, a, where $gcd(a, n) > 1$, which violates the conclusion of Fermat's little theorem. The algorithm accounts for the corner cases for $N \leq 4$.

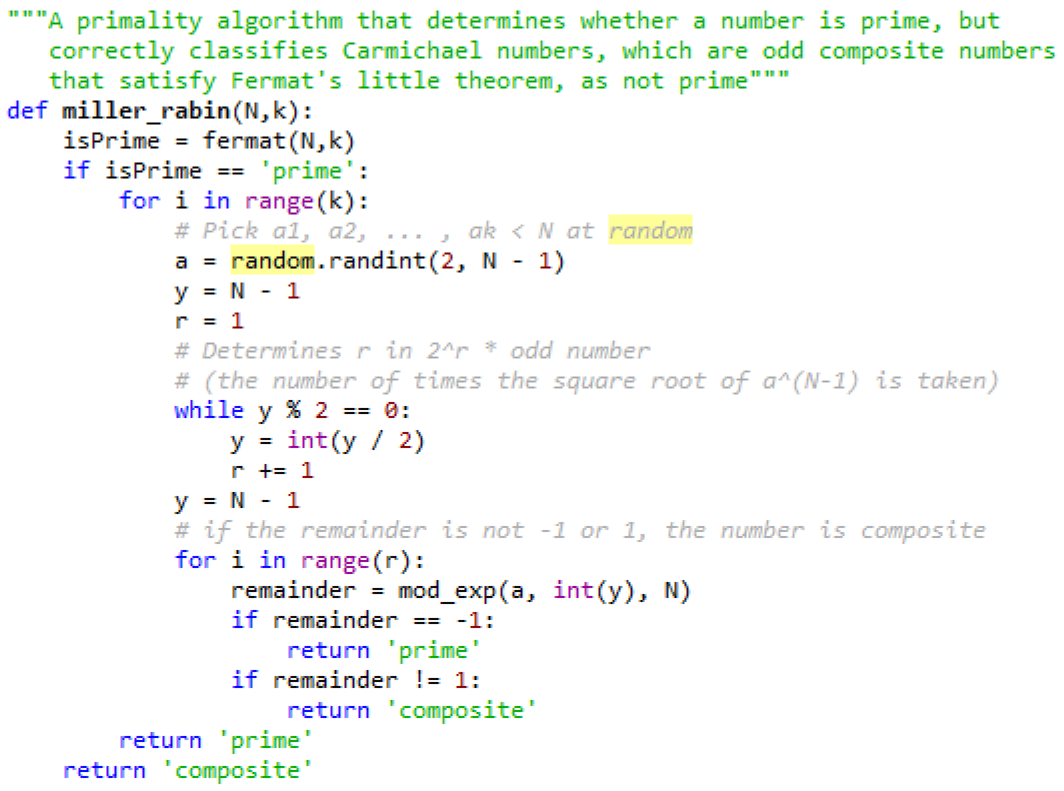


```python
"""Determines the largest positive integer that divides each of the integers"""
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)

"""Returns a randomize coprime number with N and a is in the range [2,N-1]"""
def coprime(N):
    # Pick a at random
    a = random.randint(2, N - 1)
    # if the gcd between N and a is not 1: pick another randomize a
    if gcd(N, a) != 1:
        return coprime(N)
    else:
        return a

"""A primality algorithm that determines whether a number is prime excluding
   Carmichael numbers"""
def fermat(N,k):
    # Specify the corner cases
    if N <= 1 or N == 4:
        return 'composite'
    if N <= 3:
        return 'prime'

    for i in range(k):
        # Pick a randomize coprime number 'a'
        a = coprime(N)
        # if ai^(N-1) % N does not equal 1 : composite
        if mod_exp(a, N-1, N) != 1:
            return 'composite'
    return 'prime'
```

Figure 2: Fermat Primality Algorithm.

Miller-Rabin Primality Algorithm

The Miller-Rabin algorithm states that if N is prime, $N - 1$ is an even number that can be written as $2^r * y$ where r and y are positive integers and y is odd. For each a, either $a^y \equiv 1 \bmod N$ or $a^{2 ^\wedge r * y} \equiv -1$ mod N will show that a number is prime including Carmichael numbers. For the algorithm provided, the range of a was $2 \leq a < N$. 1 was not used because 1 to the power of some number is always 1. The algorithm provided first checks whether the Fermat primality test returns ‘prime’. If ‘prime’, r is determined in 2^r * *odd number* to determine the number of iterations of taking the square root of a^{N-1}. If the remainder from the modular exponentiation results in -1, the number is prime. If the remainder does not equal 1 disregarding -1, then the number is composite. If the algorithm returns 1 as the remainder for all iterations, the number is considered prime. If the initial check from the Fermat primality test was not ‘prime’, ‘composite’ is returned.



```
"""A primality algorithm that determines whether a number is prime, but
   correctly classifies Carmichael numbers, which are odd composite numbers
   that satisfy Fermat's little theorem, as not prime"""
def miller_rabin(N,k):
    isPrime = fermat(N,k)
    if isPrime == 'prime':
        for i in range(k):
            # Pick a1, a2, ... , ak < N at random
            a = random.randint(2, N - 1)
            y = N - 1
            r = 1
            # Determines r in 2^r * odd number
            # (the number of times the square root of a^(N-1) is taken)
            while y % 2 == 0:
                y = int(y / 2)
                r += 1
            y = N - 1
            # if the remainder is not -1 or 1, the number is composite
            for i in range(r):
                remainder = mod_exp(a, int(y), N)
                if remainder == -1:
                    return 'prime'
                if remainder != 1:
                    return 'composite'
        return 'prime'
    return 'composite'
```

Figure 3: Miller Rabin Primality Algorithm.

Fermat and Miller-Rabin Probabilities

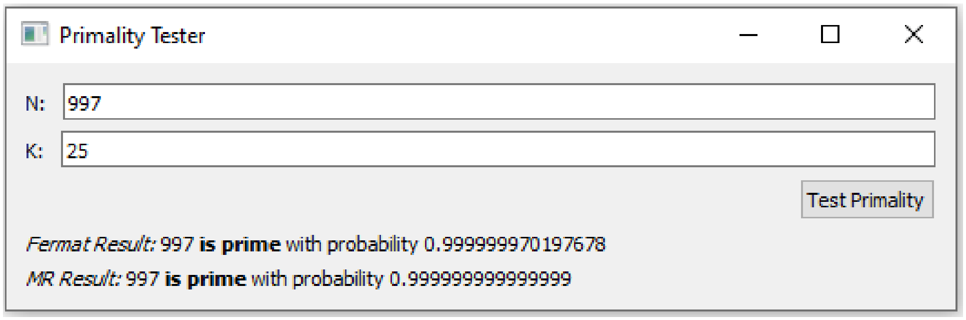
The Fermat probability proof is for every element $a < N$ that passes the Fermat primality test with respect to N, $a^{N-1} \equiv 1 \bmod N$, there is an $a * b$ that fails the Fermat primality test such that $(a * b)^{N-1} \equiv a^{N-1} * b^{N-1} \equiv b^{N-1} \not\equiv 1 \bmod N$. This results in $1 - 0.5^k$ probability of N being prime for k repetitions. Then combining the Fermat primality test with the Miller-Rabin primality test of taking the square root, the probability is reduced to $1 - 0.25^k$ of N being prime because of the inclusion of Carmichael numbers being composite.

```python
"""Determines the probability of the fermat algorithm where at least half of
   the possible values of 'a' are between 2 and N-1 are prime"""

def fprobability(k):
    return 1 - 0.5**k

"""Determines the probability of the miller rabin algorithm where at least
   one-fourth of the possible values of 'a' are between 2 and N-2 are prime"""
def mprobability(k):
    return 1 - 0.25**k
```

Figure 4: Fermat and Miller-Rabin probability of being prime.

Experimentation of Inputs

Experimentation and verification were performed on examples of N for prime numbers, composite numbers, and Carmichael numbers. The Carmichael numbers that were tested were {561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973, 75361, 101101}. These Carmichael numbers resulted in prime for the Fermat primality algorithm but composite for the Miller-Rabin primality algorithm. Some composite numbers for N were also tested including {400, 312, 998, 95, 660, 77} where both the Fermat primality algorithm and the Miller-Rabin primality algorithm concluded that N was a non-prime number. A few prime numbers for N were also checked including {41, 997, 373, 101, 571} where both the Fermat primality algorithm and the Miller-Rabin primality algorithm concluded that N was a prime number.



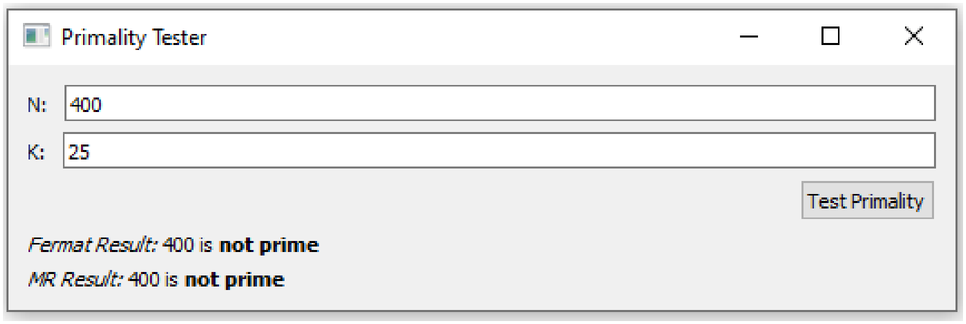
N: 997

K: 25

Fermat Result: 997 **is prime** with probability 0.999999970197678

MR Result: 997 **is prime** with probability 0.999999999999999

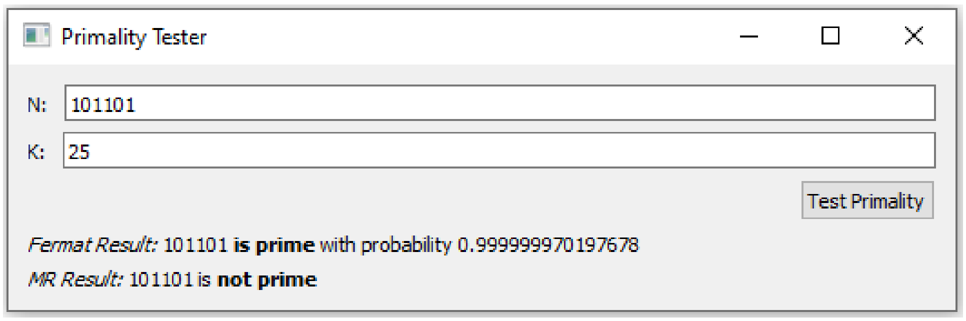
Figure 5: An example of N as a prime number.



N: 400
K: 25

Fermat Result: 400 is **not prime**
MR Result: 400 is **not prime**

Figure 6: An example of N as a non-prime, composite number.



N: 101101
K: 25

Fermat Result: 101101 **is prime** with probability 0.999999970197678
MR Result: 101101 is **not prime**

Figure 7: An example of N that is a Carmichael number.

Time and Space Complexity of Algorithms

Modular exponentiation algorithm. The iterative version of the modular exponentiation algorithm is O(n^3). Within the for loop, there is multiplication which amounts to n * (n^2). The modulo has a runtime of O(1).

Euclid algorithm. The time and space complexity for gcd is O(n^3). The base case is reached in 2n recursive calls. Each recursive call has a quadratic-time division with a runtime of n^3.

Coprime algorithm. The time and space complexity for finding a coprime number is O(n^3). The algorithm calls gcd which has a runtime of O(n^3).

Fermat primality algorithm. The algorithm's runtime is O(n^4). The algorithm is repeated k times. Modular exponentiation and the coprime algorithm both have a runtime of O(n^3) which amounts to n * (n^3).

Miller-Rabin primality algorithm. The algorithm's time and space complexity is O(n^4log(n)). The Fermat primality test utilitized in this algorithm is O(n^4). Within the for loop (n), there is a while loop that determines r in 2^r * odd number, that includes dividing by 2, amounting to n * log(n) * (n^2). There is also a for loop that iterates through the number of square roots taken from a^{N-1}, log(n). In this for loop, modular exponentiation is called which has a O(n^3) runtime, which amounts to n * log(n) * (n^3).

Fermat and Miller-Rabin probabilities. Both algorithms have a space and time complexity of O(n^k) where k is the number of repetitions.