Miller–Rabin Algorithm

1. What does the algorithm do?

Miller-Rabin is an approach to testing the primality of large numbers. The Miller-Rabin primality test, and it decides whether a number is prime or composite. It doesn't guarantee that a number is prime, but it determines the probability of a number being prime or composite.

2. How does it work?

When given a positive integer, it tries to prove that the number is not prime. If it cannot prove that the number given is not prime, then that value is probably prime. It builds on Fermat's Little Theorem, which says that if a number *n* is prime, then for any number a $1 < a < n - 1$, it must be true that $a^{n-1} \equiv mod\, n$. If this doesn't hold for some *a*, then *n* is definitely composite. However, some composite numbers like 341 can pass Fermat's test for many values of *a*, making the output appear prime. These are called Carmichael numbers, which are pseudoprimes. Due to these numbers, the Fermat test alone isn't reliable. To improve this, the Euler test looks more closely at the square roots when *a* gives a remainder of 1 when divided by *n*. For a prime number, $\frac{a^{n-1}}{2}$ should be equivalent to either 1 or -1 when divided by *n*. If it doesn't, the number is definitely composite. However, Euler's test isn't perfect either; some numbers like 341 can also pass it falsely. The Miller-Rabin test combines the ideas from both Fermat and Euler's tests and takes it further: it checks for nontrivial square roots of 1, which shouldn't exist if *n* is prime. It runs multiple rounds with different values of *a*, and if *n* fails any round, it's composite. If it passes many rounds, it's probably prime, with high confidence. The test also relies on ideas like Euclid's Lemma, which helps in understanding divisibility rules used during the algorithm's steps.

Miller-Rabin Steps:

1. $For\ some\ number\ n,\ write\ n - 1\ as\ 2^k * m,\ where\ m\ is\ odd$
   $As\ an\ example,\ let's\ do\ 104513\ as\ n$:
   $104513 - 1 = 2^6 * 1633$
   $So\ our\ k = 6\ and\ m = 1633$
2. $Pick\ a\ random\ base\ such\ that\ 1 < a < n - 1$
   $1 < a < 104512$
   $Let's\ do\ a = 3$
3. $Compute\ a^{2^{k-1}}\ mod\, n \equiv \pm 1\ mod\, n$

$$3^{1633} \, mod \, 104513 \equiv 88958$$
$$3^{2*1633} \, mod \, 104513 \equiv 88958^2 \, mod \, 104513 \equiv 10430$$
$$3^{4*1633} \, mod \, 104513 \equiv 10430^2 \, mod \, 104513 \equiv 91380$$
$$3^{8*1633} \, mod \, 104513 \equiv 91380^2 \, mod \, 104513 \equiv 29239$$
$$3^{16*1633} \, mod \, 104513 \equiv 29239^2 \, mod \, 104513 \equiv 2781$$
$$3^{32*1633} \, mod \, 104513 \equiv 2781^2 \, mod \, 104513 \equiv -1 \, mod \, 104513$$

$Since\ we\ get\ a\ result\ of\ -1\ mod\ 104513\ then\ our\ n\ is\ probably\ prime$

3. What can it be used for?

The Miller-Rabin test is used in cryptography, blockchain systems, random prime generation, and secure hashing or digital signatures. Encryption algorithms like RSA, Diffie-Hellman, and ElGamal rely on very large prime numbers to generate encryption keys that keep information secure. The Miller-Rabin test provides a fast and reliable way of checking whether a number is probably prime. Compared to slower methods like trial division or factoring, which are impractical for large numbers, Miller-Rabin can quickly determine with high confidence whether a number is probably prime. Its probabilistic approach, when repeated with multiple random values, makes the chance of a false positive extremely small. The test is also easy to implement in code, as a result, it is commonly built into most programming libraries like SymPy in Python and OpenSSL in C.

4. How efficient is it?

The Miller–Rabin test is very efficient, especially for large numbers:

- Time complexity: $O(k * log^3 n)$, where:
  - n is the number being tested
  - k is the number of test rounds
- With enough rounds, the probability of error becomes extremely low, making it reliable for practical purposes.
  - chance of being wrong 4^{-n}, n is random number of values

The Algorithm (Python):

```
import random
```

```
# Utility function to do modular exponentiation.
# It returns (x^y) % p
def power(x, y, p):

    # Initialize result
    res = 1;

    # Update x if it is more than or equal to p
    x = x % p;
    while (y > 0):

        # If y is odd, multiply x with result
        if (y & 1):
            res = (res * x) % p;

        # y must be even now
        y = y>>1; # y = y/2
        x = (x * x) % p;

    return res;

# This function is called for all k trials. It returns false if n is composite and returns false if n is probably prime. m is an odd
number such that m*2<sup>k</sup> = n-1 for some k >= 1
def millerTest(m, n):

    # Pick a random number in [2..n-2]
    # Corner cases make sure that n > 4
    a = 2 + random.randint(1, n - 4);

    # Compute a^m % n
    x = power(a, m, n);

    if (x == 1 or x == n - 1):
        return True;

    # Keep squaring x while one of the following doesn't happen
    # (i) m does not reach n-1
    # (ii) (x^2) % n is not 1
    # (iii) (x^2) % n is not n-1
    while (m != n - 1):
        x = (x * x) % n;
        m *= 2;

        if (x == 1):
            return False;
        if (x == n - 1):
            return True;

    # Return composite
    return False;
```

```
# It returns false if n is composite and returns true if n is probably prime. r is an input parameter that determines the accuracy level. Higher value of r indicates more accuracy.
def isPrime( n, r):

    # Corner cases
    if (n <= 1 or n == 4):
        return False;
    if (n <= 3):
        return True;

    # Find r such that n = 2^d * k + 1 for some k >= 1
    m = n - 1;
    while (m % 2 == 0):
        m //= 2;

    # Iterate given number of 'r' times
    for i in range(r):
        if (millerTest(m, n) == False):
            return False;

    return True;

# Driver Code
# Number of iterations
r = 4;

print("All primes smaller than 100: ");
for n in range(1,100):
    if (isPrime(n, r)):
        print(n , end=" ");
```

An Example Output:

Input: n = 13, k = 2.
1) Compute d and r such that d*2r = n-1,
   d = 3, r = 2.
2) Call millerTest k times.

- Step 1:

1) Pick a random number *a* in the range [2, n-2]
   Suppose a = 4
2) Compute: x = pow(a, d) % n
   x = 43 % 13 = 12
3) Since x = (n-1), return true.

- Step 2:

1) Pick a random number *a* in the range [2, n-2]

Suppose a = 5
2) Compute: x = pow(a, d) % n
x = 53 % 13 = 8
3) x neither 1 nor 12.
4) Do the following (r-1) = 1 times
a) x = (x * x) % 13 = (8 * 8) % 13 = 12
b) Since x = (n-1), return true.
Since both iterations return true, we return true.

Citations

Boneh, Dan. "Miller-Rabin Primality Test." *Stanford University: Crypto Notes*, https://crypto.stanford.edu/pbc/notes/numbertheory/millerrabin.html.

Conrad, Keith. *The Miller-Rabin Primality Test*. University of Connecticut, https://kconrad.math.uconn.edu/blurbs/ugradnumthy/millerrabin.pdf.

GeeksforGeeks. "Euclid's Lemma." *GeeksforGeeks*, https://www.geeksforgeeks.org/euclids-lemma/.

GeeksforGeeks. "Fermat's Little Theorem." *GeeksforGeeks*, https://www.geeksforgeeks.org/fermats-little-theorem/.

GeeksforGeeks. "Primality Test | Set 3 (Miller–Rabin)." *GeeksforGeeks*, 12 Sept. 2023, https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/.

Hoffoss, Douglas. *The Rabin-Miller Test*. University of San Diego, https://home.sandiego.edu/~dhoffoss/teaching/cryptography/10-Rabin-Miller.pdf.

Sen, Gaurav. *Miller-Rabin Primality Test – The Fastest Way to Check for Primes*. YouTube, 11 Aug. 2020, https://www.youtube.com/watch?v=zmhUlVck3J0.

TutorialsPoint. "What Are the Miller Rabin Algorithm for Testing the Primality of a Given Number?" *TutorialsPoint*, https://www.tutorialspoint.com/what-are-the-miller-rabin-algorithm-for-testing-the-primality-of-a-given-number.