Primality Test | Set 3 (Miller–Rabin) - GeeksforGeeks

Source: https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/

 Primality Test | Set 3 (Miller–Rabin) Last Updated : 23 Jul, 2025 Given a number n, check if it is prime or not. We have introduced and discussed School and Fermat methods for primality testing. Primality Test | Set 1 (Introduction and School Method) Primality Test | Set 2 (Fermat Method) In this post, the Miller-Rabin method is discussed. This method is a probabilistic method (like Fermat), but it is generally preferred over Fermat's method. Algorithm: // It returns false if n is composite and returns true if n // is probably prime. k is an input parameter that determines // accuracy level. Higher value of k indicates more accuracy. bool isPrime(int n, int k) 1) Handle base cases for n < 3 2) If n is even, return false. 3) Find an odd number d such that n-1 can be written as d*2 r . Note that since n is odd, (n-1) must be even and r must be greater than 0. 4) Do following k times if (millerTest(n, d) == false) return false 5) Return true.

// This function is called for all k trials. It returns // false if n is composite and returns true if n is probably // prime. // d is an odd number such that d*2 r = n-1 for some r>=1 bool millerTest(int n, int d) 1) Pick a random number 'a' in range [2, n-2] 2) Compute: x = pow(a, d) % n 3) If x == 1 or x == n-1, return true.

// Below loop mainly runs 'r-1' times. 4) Do following while d doesn't become n-1. a) x = (x*x) % n. b) If (x == 1) return false. c) If (x == n-1) return true. Example: Input: n = 13, k = 2.

1) Compute d and r such that d*2 r = n-1, d = 3, r = 2. 2) Call millerTest k times. 1 st Iteration: 1) Pick a random number 'a' in range [2, n-2] Suppose a = 4

2) Compute: x = pow(a, d) % n x = 4 3 % 13 = 12

3) Since x = (n-1), return true. II nd Iteration: 1) Pick a random number 'a' in range [2, n-2] Suppose a = 5

2) Compute: x = pow(a, d) % n x = 5 3 % 13 = 8

3) x neither 1 nor 12.

4) Do 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. Recommended: Please solve it on " PRACTICE " first, before moving on to the solution. Implementation: Below is the implementation of the above algorithm. C++ // C++ program Miller-Rabin primality test #include <bits/stdc++.h> using namespace std ; // Utility function to do modular exponentiation. // It returns (x^y) % p int power (int x , unsigned int y , int p) { int res = 1 ; // Initialize result x = x % p ; // Update x if it is more than or // equal to 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 true if n is // probably prime. // d is an odd number such that d*2 = n-1 // for some r >= 1 bool miillerTest (int d , int n) { // Pick a random number in [2..n-2] // Corner cases make sure that n > 4 int a = 2 + rand () % (n - 4); // Compute a^d % n int x = power (a , d , n); if (x == 1 || x == n -1) return true ; // Keep squaring x while one of the following doesn't // happen // (i) d does not reach n-1 // (ii) (x^2) % n is not 1 // (iii) (x^2) % n is not n-1 while (d != n -1) { x = (x * x) % n ; d *= 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. k is an input parameter that determines // accuracy level. Higher value of k indicates more accuracy. bool isPrime (int n , int k) { // Corner cases if (n <= 1 || n == 4) return false ; if (n <= 3) return true ; // Find r such that n = 2^d * r + 1 for some r >= 1 int d = n - 1 ; while (d % 2 == 0) d /= 2 ; // Iterate given number of 'k' times for (int i = 0 ; i < k ; i ++) if (! miillerTest (d , n)) return false ; return true ; } // Driver program int main () { int k = 4 ; // Number of iterations cout << "All primes smaller than 100: \n " ; for (int n = 1 ; n < 100 ; n ++) if (isPrime (n , k)) cout << n << " " ; return 0 ; } Java // Java program Miller-Rabin primality test import java.io.* ; import java.math.* ; class GFG { // Utility function to do modular // exponentiation. It returns (x^y) % p static int power (int x , int y , int p) { int res = 1 ; // Initialize result //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) == 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. // d is an odd number such that d*2^r // = n-1 for some r >= 1 static boolean miillerTest (int d , int n) { // Pick a random number in [2..n-2] // Corner cases make sure that n > 4 int a = 2 + (int)(Math . random () % (n - 4)); // Compute a^d % n int x = power (a , d , n); if (x == 1 || x == n - 1) return true ; // Keep squaring x while one of the // following doesn't happen // (i) d does not reach n-1 // (ii) (x^2) % n is not 1 // (iii) (x^2) % n is not n-1 while (d != n - 1) { x = (x * x) % n ; d *= 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. k is an input parameter that // determines accuracy level. Higher // value of k indicates more accuracy. static boolean isPrime (int n , int k) { // Corner cases if (n <= 1 || n == 4) return false ; if (n <= 3) return true ; // Find r such that n = 2^d * r + 1 // for some r >= 1 int d = n - 1 ; while (d % 2 == 0) d /= 2 ; // Iterate given number of 'k' times for (int i = 0 ; i < k ; i ++) if (! miillerTest (d , n)) return false ; return true ; } // Driver program public static void main (String args []) { int k = 4 ; // Number of iterations System . out . println ("All primes smaller " + "than 100: "); for (int n = 1 ; n < 100 ; n ++) if (isPrime (n , k)) System . out . print (n + " "); } } /* This code is contributed by Nikita Tiwari.*/ Python3 # Python3 program Miller-Rabin primality test 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. d is an odd # number such that d*2^r = n-1 # for some r >= 1 def miillerTest (d , 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^d % n x = power (a , d , n); if (x == 1 or x == n - 1): return True ; # Keep squaring x while one # of the following doesn't # happen # (i) d does not reach n-1 # (ii) (x^2) % n is not 1 # (iii) (x^2) % n is not n-1 while (d != n - 1): x = (x * x) % n ; d *= 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. k is an # input parameter that determines # accuracy level. Higher value of # k indicates more accuracy. def isPrime (n , k): # Corner cases if (n <= 1 or n == 4): return False ; if (n <= 3): return True ; # Find r such that n = # 2^d * r + 1 for some r >= 1 d = n - 1 ; while (d % 2 == 0): d //= 2 ; # Iterate given number of 'k' times for i in range (k): if (miillerTest (d , n) == False): return False ; return True ; # Driver Code # Number of iterations k = 4 ; print ("All primes smaller than 100: "); for n in range (1 , 100): if (isPrime (n , k)): print (n , end = " "); # This code is contributed by mits C# // C# program Miller-Rabin primality test using System ; class GFG { // Utility function to do modular // exponentiation. It returns (x^y) % p static int power (int x , int y , int p) { int res = 1 ; // Initialize result // 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) == 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. // d is an odd number such that d*2^r // = n-1 for some r >= 1 static bool miillerTest (int d , int n) { // Pick a random number in [2..n-2] // Corner cases make sure that n > 4 Random r = new Random (); int a = 2 + (int)(r . Next () % (n - 4)); // Compute a^d % n int x = power (a , d , n); if (x == 1 || x == n - 1) return true ; // Keep squaring x while one of the // following doesn't happen // (i) d does not reach n-1 // (ii) (x^2) % n is not 1 // (iii) (x^2) % n is not n-1 while (d != n - 1) { x = (x * x) % n ; d *= 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. k is an input parameter that // determines accuracy level. Higher // value of k indicates more accuracy. static bool isPrime (int n , int k) { // Corner cases if (n <= 1 || n == 4) return false ; if (n <= 3) return true ; // Find r such that n = 2^d * r + 1 // for some r >= 1 int d = n - 1 ; while (d % 2 == 0) d /= 2 ; // Iterate given number of 'k' times for (int i = 0 ; i < k ; i ++) if (miillerTest (d , n) == false) return false ; return true ; } // Driver Code static void Main () { int k = 4 ; // Number of iterations Console . WriteLine ("All primes smaller " + "than 100: "); for (int n = 1 ; n < 100 ; n ++) if (isPrime (n , k)) Console . Write (n + " "); } } // This code is contributed by mits PHP <?php // PHP program Miller-Rabin primality test // Utility function to do // modular exponentiation. // It returns (x^y) % p function 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 $res ; } // This function is called // for all k trials. It returns // false if n is composite and // returns false if n is // probably prime. d is an odd // number such that d*2^r = n-1 // for some r >= 1 function miillerTest ($d , $n) { // Pick a random number in [2..n-2] // Corner cases make sure that n > 4 $a = 2 + rand () % ($n - 4); // Compute a^d % n $x = power ($a , $d , $n); if ($x == 1 || $x == $n - 1) return true ; // Keep squaring x while one // of the following doesn't // happen // (i) d does not reach n-1 // (ii) (x^2) % n is not 1 // (iii) (x^2) % n is not n-1 while ($d != $n - 1) { $x = ($x * $x) % $n ; $d *= 2 ; if ($x == 1) return false ; if ($x == $n - 1) return true ; } // Return composite return false ; } // It returns false if n is // composite and returns true if n // is probably prime. k is an // input parameter that determines // accuracy level. Higher value of // k indicates more accuracy. function isPrime ($n , $k) { // Corner cases if ($n <= 1 || $n == 4) return false ; if ($n <= 3) return true ; // Find r such that n = // 2^d * r + 1 for some r >= 1 $d = $n - 1 ; while ($d % 2 == 0) $d /= 2 ; // Iterate given number of 'k' times for ($i = 0 ; $i < $k ; $i ++) if (! miillerTest ($d , $n)) return false ; return true ; } // Driver Code // Number of iterations $k = 4 ; echo "All primes smaller than 100: \n " ; for ($n = 1 ; $n < 100 ; $n ++) if (isPrime ($n , $k)) echo $n , " " ; // This code is contributed by ajit ?> JavaScript < script > // Javascript program Miller-Rabin primality test // Utility function to do // modular exponentiation. // It returns (x^y) % p function power (x , y , p) { // Initialize result let 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. d is an odd // number such that d*2^r = n-1 // for some r >= 1 function miillerTest (d , n) { // Pick a random number in [2..n-2] // Corner cases make sure that n > 4 let a = 2 + Math . floor (Math . random () * (n - 2)) % (n - 4); // Compute a^d % n let x = power (a , d , n); if (x == 1 || x == n - 1) return true ; // Keep squaring x while one // of the following doesn't // happen // (i) d does not reach n-1 // (ii) (x^2) % n is not 1 // (iii) (x^2) % n is not n-1 while (d != n - 1) { x = (x * x) % n ; d *= 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. k is an // input parameter that determines // accuracy level. Higher value of // k indicates more accuracy. function isPrime (n , k) { // Corner cases if (n <= 1 || n == 4) return false ; if (n <= 3) return true ; // Find r such that n = // 2^d * r + 1 for some r >= 1 let d = n - 1 ; while (d % 2 == 0) d /= 2 ; // Iterate given number of 'k' times for (let i = 0 ; i < k ; i ++) if (! miillerTest (d , n)) return false ; return true ; } // Driver Code // Number of iterations let k = 4 ; document . write ("All primes smaller than 100:
"); for (let n = 1 ; n < 100 ; n ++) if (isPrime (n , k)) document . write (n , " "); // This code is contributed by gfgking < /script> Output: All primes smaller than 100: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 Time Complexity: O(k*logn) Auxiliary Space: O(1) How does this work? Below are some important facts behind the algorithm: Fermat's theorem states that, If n is a prime number, then for every a, 1 <= a < n, a n-1 % n = 1 Base cases make sure that n must be odd. Since n is odd, n-1 must be even. And an even number can be written as d * 2 s where d is an odd number and s > 0. From the above two points, for every randomly picked number in the range [2, n-2], the value of a d*2r % n must be 1. As per Euclid's Lemma , if x 2 % n = 1 or (x 2 - 1) % n = 0 or (x-1)(x+1)% n = 0. Then, for n to be prime, either n divides (x-1) or n divides (x+1). Which means either x % n = 1 or x % n = -1. From points 2 and 3, we can conclude For n to be prime, either a d % n = 1 OR a d*2i % n = -1 for some i, where 0 <= i <= r-1. Next Article : Primality Test | Set 4 (Solovay-Strassen) This article is contributed Ruchir Garg . Comment Article Tags: Article Tags: Mathematical DSA Modular Arithmetic Prime Number number-theory + 1 More