Miller-Rabin Primality Test

Miller-Rabin primality test is an optimization of Fermat primality test (based on Fermat’s little theorem).

Full Implementation Codes:

**import** random

"""

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

"""

**def** millerTest**(**d**,** n**):**

# Pick a random number in [2, n-2]

# Corner cases in isPrime function make sure that n > 4

a **=** 2 **+** random**.**randint**(**1**,** n **-** 4**)**

# Compute a ^ d % n

x **=** **pow(**a**,** d**,** n**)**

**if** **(**x **==** 1 **or** x **==** n **-** 1**):**

**return** **True**

"""

Keep squaring x while one of the following does not happen

(1) d does not reach n - 1

(2) (x ^ 2) % n is not 1

(3) (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**

# If no x satisfies, n is a composite

**return** **False**

"""

It returns False if n is composite and returns True if n is probably prime (pseudoprime).

k is an input parameter that determines accuracy level. Higher level 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 ^ s \* d + 1 for some d >= 1

# d is an odd number

d **=** n **-** 1

**while** **(**d **%** 2 **==** 0**):**

d **//=** 2

# Iterate given number of "k" times

**for** i **in** **range(**k**):**

**if** **(**millerTest**(**d**,** n**)** **==** **False):**

**return** **False**

**return** **True**

"""

Main programme

"""

**def** main**():**

**#** Number of iterations

k **=** 4

upperBound **=** **int(input(**"Find all primes below: "**))**

**print(**f"All primes smaller than {upperBound}: "**)**

**print()**

counter **=** 0

**for** n **in** **range(**1**,** upperBound**):**

**if** **(**isPrime**(**n**,** k**)):**

**print(**n**,** end**=**" "**)**

counter **+=** 1

**print(**"\n"**)**

**print(**f"{counter} primes in total"**)**

**print(**"\n" **\*** 3**)**

main**()**

Mathematical Theories:



Back to the Python Codes:

1. Firstly, let d = n -1. Keep taking out 2 from d, until d is an odd number (to fulfill the format of [1]).

# Find r such that n = 2 ^ s \* d + 1 for some d >= 1

# d is an odd number

d **=** n **-** 1

**while** **(**d **%** 2 **==** 0**):**

d **//=** 2

1. Specify the number of iterations. The Miller-Rabin primality test can find pseudoprimes, which means there is a possibility that the number found is a composite. Repeating the algorithm with different random value of a will increase the accuracy, but take longer time.

**#** Number of iterations

k **=** 4

1. Generate random number a according to the requirement in Fermat’s little theorem

# Pick a random number in [2, n-2]

# Corner cases in isPrime function make sure that n > 4

a **=** 2 **+** random**.**randint**(**1**,** n **-** 4**)**

1. Verify if or .



# Compute a ^ d % n

x **=** **pow(**a**,** d**,** n**)**

**if** **(**x **==** 1 **or** x **==** n **-** 1**):**

**return** **True**

1. If first 2 cases not satisfied, keep trying the rest, until d = n – 1 (the case ).



**while** **(**d **!=** n **-** 1**):**

x **=** **(**x **\*** x**)** **%** n

d **\*=** 2

**if** **(**x **==** 1**):**

**return** **False**

**if** **(**x **==** n **-** 1**):**

**return** **True**

# If no x satisfies, n is a composite

**return** **False**

Running Outcome:

文本

描述已自动生成

Although “pseudoprime” sounds not very accurate, the result is actually very reliable. In the case above, k is set to 4.