#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 false if n is

// probably prime.

// d is an odd number such that d\*2<sup>r</sup> = 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 nber 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;

}

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\*2r.

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 false if n is probably

// prime.

// d is an odd number such that d\*2r = 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\*2r = n-1,

d = 3, r = 2.

2) Call millerTest k times.

1st Iteration:

1) Pick a random number 'a' in 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.

IInd Iteration:

1) Pick a random number 'a' in 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 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.