“STUDY OF PRIMALITY TESTS”

PROJECT REPORT

Submitted for CAL in B.Tech Data Structures and Algorithms(CSE2003)

By

CHRIS JOY KOKKAT (17BCE1271) &
AKSHAY LAXMIKANT KULKARNI(17BCE1115)

Slot:E2

Name of faculty: DR. SHRIDEVI S

(SCHOOL OF COMPUTER SCIENCE AND ENGINEERING)

VIT® UNIVERSITY (Estd. u/s 3 of UGC Act 1956) VELLORE ▪ CHENNAI www.vit.ac.in

March, 2018

<u>CERTIFICATE</u>

This is to certify that the Project work entitled "**PRIMALITY TESTS**" that is being submitted by "**Chris Joy Kokkat & Akshay Laxmikant Kulkarni**" for CAL in BTech Data Structures & Algorithms(CSE2003) is a record of bonafide work done under my supervision. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted for any other CAL course.

Place : Chennai

Date : 01/04/18

Signature of Students: Chris Joy Kokkat & Akshay Laxmikant Kulkarni

Signature of Faculty: Dr. Shridevi S

ACKNOWLEDGEMENTS

We would like to thank my professor for the course, Dr. Shridevi S, for guiding us and helping us complete my project.

We would also like to thank the management for giving us a chance to carry out our studies at the University.

Chris Joy Kokkat
Reg. No. 17BCE1271

Akshay Laxmikant Kulkarni
Reg. No. 17BCE1115

ABSTRACT

A prime number is a natural number greater than 1 that has no factors other than 1 and the number itself. Numbers that are not prime are called composite numbers.

Prime numbers play an important role in cryptography. Many popular algorithms used in public-key cryptography are based on the fact that integer factorization is a very hard problem, that is, the time required to convert a number into its prime factors grows exponentially with the number of bits used by the number in storage. Prime numbers also play a key role in number theory and quantum mechanics.

In this project, a study will be done on the existing tests that check whether a number is prime or not, otherwise called Primality Tests. This will include the properties of prime numbers and the tests.

1. INTRODUCTION

The following Primality Tests were studied:

1)Trial Division
2)Fermat's Primality Test
3)Miller-Rabin Test
4)AKS Primality Test

In the following pages, there will be an explanation of the test, its advantages and disadvantages and the source code in C or C++ along with their time complexities.

Usually, Primality Tests are used to check odd natural numbers, since the only even prime number is 2. All methods have this in consideration.

One point to note would be that all the programs were run on a computer that has the following specifications:

- Processor: Intel™ Core i5 6th Gen.
- RAM: 4GB
- 64-bit Windows 10 OS

The sample input for all the algorithms is: 1000000007 which is a Prime Number.

2. TRIAL DIVISION

This method is the simplest and the most basic of Primality Tests.

2.1. METHOD

Assume the number to be tested as *n*. The method divides *n* with another number less than *n* and checks whether *n* is divisible by it or not. This other number, say *a*, can range from 2 to *n*. If a case occurs where *a* fully divides *n*, the method returns composite. If all values of *a* are tested with no case of divisibility, the method returns prime.

2.2. SOURCE CODE

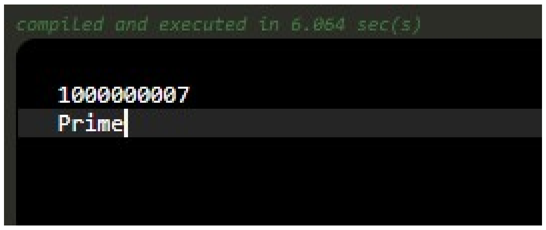
```
#include <bits/stdc++.h>
using namespace std;

bool isPrime(unsigned int n){
        for(int i =2;i < n; i++){
                if(n % i == 0) //If remainder when n is divided by i, is zero
                        return false;
        }
        return true;
}
```

TIME COMPLEXITY: O(n)

2.3. PROBLEMS

This method is fairly simple to understand and code. However, it is very laborious and slow.

2.4. OUTPUT



```
compiled and executed in 6.064 sec(s)

   1000000007
   Prime
```

Time taken to compile and execute : 6.064 sec

3. FERMAT'S PRIMALITY TEST

This method is the precursor to the Miller-Rabin Test and the AKS Primality Test. It is derived from Fermat's Little Theorem.

3.1. FERMAT'S LITTLE THEOREM

Fermat's Little Theorem is a fundamental theorem in elementary number theory. It states that if *p* is a prime number, then for any integer *a*, $a^p - a$ is divisible by *p*. In notation of modular arithmetic, this can be written as

$$a^p \equiv a (mod\ p)$$ – Equation 3.1

If *a* is not divisible by *p*, Fermat's Little Theorem can be written as

$$a^p \equiv 1 (mod\ p)$$ – Equation 3.2

This theorem is the basis to all the recent primality tests, like Miller-Rabin and AKS.

3.2. METHOD

Fermat’s Primality Test is a probabilistic test. This means that it cannot fully confirm the primality of a number. If a number *n* is to be tested, we pick a random number that is not divisible by *n*, say *a* and see whether equation 3. 2 holds. If it holds true, we can say that *n* is probably prime. In this case, we try again with another value for *a*. The more iterations we use, the better is the probability for the confirmation of primality of *n*.

If the equation does not hold true for a value of *a*, we can say that *n* is composite.

There are certain terminologies that must be mentioned.

If *n* is a composite number that satisfies equation to for some given values of *a*, then *n* is called a *Fermat pseudoprime* to the base *a*. If there exists an *a** that does not satisfy the equation, it is called the *Fermat witness* to the compositeness of *n*, and the other values of *a* for which it held true are called *Fermat liars*.

Usually *a* is chosen within the range of [2,n-2].

3.3. SOURCE CODE

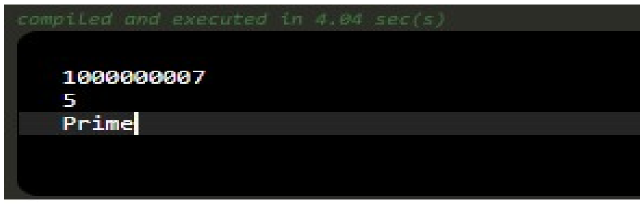
```
#include <bits/stdc++.h>
using namespace std;

/*This function returns (x^y) modulo p*/
long int modpow(int x, unsigned int y, int p)
{
   long int r0 = 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)
         r0 = (r0*x) % p;

      // y must be even now
      y /= 2; // y = y/2
      x = (x*x) % p;
   }
   return r0;
}


bool isPrime(unsigned int n, int k) //k is the number of iterations
{
  if (n <= 1 || n == 4)  return false;
/*n ==4 is tested in corner case so that no error occurs in the random number generator
a few lines ahead*/
  if (n <= 3) return true;

  while (k>0)
  {
     int a = 2 + rand()%(n-4);  //this expression returns a random number in range of [2,
n-2]
     if (modpow(a, n-1, n) != 1)
        return false;

     k--;
```

```
    }
 return true;}
```
TIME COMPLEXITY : O(k.log^2(n).log(log(n)).log(log(log(n))))

3.4. PROBLEMS

The main problem with this test is that there are infinitely many *Fermat pseudoprimes*. Also, there are infinitely many *Carmichael numbers*. These are numbers *n* for which all values of *a* that are not divisible by *n*, satisfy the Fermat's Little Theorem. The amount of Carmichael numbers are enough that the Fermat's Primality Test is not used in this form.

3.5. OUTPUT



```
compiled and executed in 4.04 sec(s)

   1000000007
   5
   Prime
```

Time taken to compile and execute : 4.04 sec.

4. MILLER-RABIN PRIMALITY TEST

The Miller-Rabin Test is a probabilistic test. It uses the Fermat's Little Theorem. It was introduced by Gary L. Miller in 1976 as a deterministic test whose correctness relied on the unproven extend Riemann hypothesis. In 1980, Michael O. Rabin modified it to create a probabilistic algorithm.

4.1. METHOD

Suppose the number to be checked is *n*. We can write *n* in the following way.

$n=2^s \cdot d+1$ – Equation 4.1

where *s* and *d* are positive integers, and *d* is odd.

Now, there is a lemma which states that 1 and -1 give the result 1 when squared modulo *p*, where *p* is a prime number greater than 2. To show this, let *x* be a square root of 1 modulo *p*.

$x^2 \equiv 1 \left(mod\ p\right)$ – Equation 4.2

This implies

$\left(x-1\right)\left(x+1\right) \equiv 0 \left(mod\ p\right)$ – Equation 4.3

Or we can say that *p* divides *(x – 1)(x + 1)*. It divides one of the two factors. So we can say that *x* is equivalent to either 1 or -1 modulo *p*.
Now, back to our number *n*. If *n* is prime, we find values of *s* and *d*, as per equation 1.
Now, for a randomly generated *a*,

$$a^d \equiv 1 \left(mod\, n\right) \text{ – Equation 4.4}$$

Or,

$$a^{2^r \cdot d} \equiv -1 \left(mod\, n\right) \text{ – Equation 4.5}$$

Where *r* is an integer and $0 \le r \le s-1$.

To prove these statements to be true,

$$a^{n-1} \equiv 1 \left(mod\, n\right) \text{ (Equation 3.2)}$$

If we keep taking the square root of a^{n-1}, we will get either 1 or -1. If we get -1. Second equality holds true. If we never get -1, then in the end of removing ever power of 2, we are left with first equality.

So, in our function, we look for an *a* that does not satisfy the two equations to prove the compositeness of *n*. The probability of *n* being prime if given by the function is 4^{-k} Where *k* is the number of iterations.

4.2. SOURCE CODE

```
#include <bits/stdc++.h>
using namespace std;

//pow returns (x^y) % p
long int modpow(int x, unsigned int y, int p)
{
  long int r0 = 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)
      r0 = (r0*x) % p;

    // y must be even now
    y /= 2; // y = y/2
    x = (x*x) % p;
```

```
   }
   return r0;
}

bool MillerRabin(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);
 //cout<<"a: "<<a<<endl;
   // Compute a^d % n
   long int x = modpow(a, d, n);
  //  cout<<"x: "<<x<<endl;
   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;
   //    cout<<"x:"<<x<<endl<<"d:"<<d<<endl;
      if (x == 1)      return false;
      if (x == n-1)    return true;
   }

   // Return composite
   return false;
}

// Higher value of k gives more accuracy
bool isPrime(int n, int k)
{
   // Corner cases
   if (n <= 1 || n == 4)  return false;
   if (n <= 3) return true;
```

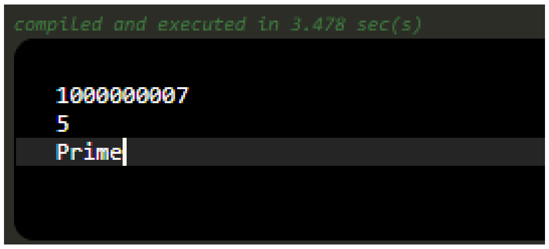
```
    // Find r such that n = 2^d * r + 1 for some r >= 1
    int d = n - 1;
    while (d % 2 == 0)
       d /= 2;
    //cout<<"d: "<<d<<endl;
    // Iterate given number of 'k' times
    for (int i = 0; i < k; i++){
      // cout<<"Iteration "<<i<<endl;
        if (MillerRabin(d, n) == false)
            return false;}

    return true;
}
```

TIME COMPLEXITY : $O(k.\log^2(n).\log(\log(n)).\log(\log(\log(n))))$

4.3. PROBLEMS

The above code can only detect prime numbers upto 47569.

4.4 OUTPUT



```
compiled and executed in 3.478 sec(s)

1000000007
5
Prime
```

Time taken to compile and execute : 3.478 sec

5. AKS PRIMALITY

The AKS Primality Test, a deterministic primality test was developed in 2002 by Manindra Agarwal, Neeraj Kayal and Nitin Saxena. This test was the first algorithm to check whether a number is prime or not, within polynomial time.

5.1. METHOD

This test works on the following theorem: If there exist an integer *n* (≥ 2¿ and integer *a* coprime to *n*, *n* is prime if and only if

$$(x+a)^n \equiv (x^n+a)(mod\,n)$$ – Equation 5.1

Where *x* is a formal symbol.

This is a generalization of Fermat's Little Theorem on polynomials. However, calculating all the coefficients of $(x+a)^n$ would take exponential time. The AKS, however evaluates the polynomial by finding the remainder after dividing it with (x^r-1) thus making computational time depend on *r*. Therefore the new relation is

$$(x+a)^n \equiv (x^n+a)(mod\, x^r-1, n) \text{ – Equation 5.2}$$

Which is the same as:

$$(x+a)^n-(x^n+a)=(x^r-1)g+nf$$

Where *g* and *f* are some polynomials.

5.2. SOURCE CODE

```
//This is just a minor implementation of the test, this is because AKS is not completely //
implemented. This implementation can be used to find primes till 63 only.
#include <stdio.h>
#include <stdlib.h>

long long c[100];

void coef(int n)
{
        int i, j;

        if (n < 0 || n > 63) abort(); // gracefully deal with range issue

        for (c[i=0] = 1; i < n; c[0] = -c[0], i++)
                for (c[1 + (j=i)] = 1; j > 0; j--)
                        c[j] = c[j-1] - c[j];
}

int is_prime(int n)
{
        int i;

        coef(n);
        c[0] += 1, c[i=n] -= 1;
        while (i-- && !(c[i] % n));

        return i < 0;
}
```

```
void show(int n)
{
        do printf("%+lldx^%d", c[n], n); while (n--);
}

int main(void)
{
        int n;

        for (n = 0; n < 10; n++) {
                coef(n);
                printf("(x-1)^%d = ", n);
                show(n);
                putchar('\n');
        }

        printf("\nprimes");
        for (n = 1; n <= 63; n++)
                if (is_prime(n))
                        printf(" %d", n);

        putchar('\n');
        return 0;
}
```

TIMECOMPLIXITY : O($\log^{15/2}$ (n))

5.3. PROBLEMS

While it is deterministic, AKS is also one of the slowest, in some cases, even slower than Trial Division. So, usually, Miller-Rabin Test is the most preferred test out of the 4.

Also, the test has never been fully implemented in code due to the difficulties in implementing certain functions required for the test. Many tests with modifications of the AKS exist. The main breakthrough of the algorithm is not the practical code that can be implemented, it is theoretical, it shows us that a polynomial time deterministic algorithm to check for prime numbers exists.

6. CONCLUSION

As of January 2018, the largest known Prime number has 23249425 decimal digits. Algorithms like these along with high processing speeds achieved using supercomputers are used to find these huge prime numbers. Since Prime Numbers are such an important part of Number Theory and Cryptography, these algorithms are very helpful in finding patterns using primes easily. The AKS test was a revolutionary breakthrough since it showed us that a deterministic test which has polynomial time complexity exists and is efficient, at least in theory. The most popular test is the Miller Rabin test which, even though it is a probabilistic algorithm, is the most efficient among all of the tests studied.

The time taken to generate output by the different tests discussed for a single output is tabulated below:

Primality Test	**Time Taken**
1. Trial Division	6.064s
2. Fermat's Test	4.04s
3. Miller Rabin Test	3.478s

7. REFERENCES

https://brilliant.org/wiki/fermats-little-theorem/

Manindra Agrawal, Neeraj Kayal & Nitin Saxena (2002). Primes is in P, Indian Institute of Technology, Kanpur

https://en.wikipedia.org/wiki/Fermat%27s_little_theorem

https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test

https://comeoncodeon.wordpress.com/2010/09/18/miller-rabin-primality-test/