Contrast various tests for primality

Manish Sharma
Department of Computer Science and Engineering
Indian Institute of Technology
Guwahati, India

Abstract

Prime Numbers have immense use in cryptography and network security applications. The security of public-key encryption system depends on the key fact of difficulty in generating prime factors. They have also been used in random nuber generation. In this paper, we compare various tests of primality testing, both *probabilistic* and *deterministic*. We also compare various methods on *generating* prime numbers. We also look on their uses.

I. Introduction

Prime Numbers have been long studied since past. The techniques of generating primes using sieves was long known. Many probabilistic and deterministic methods have been developed for testing the primality of a number. Many of the probabilistic tests may be deterministic if the Extended Riemann Hypothesis conjecture[1] is true.

In this paper, we compare these results when they are implemented and used practically. Large amount of theoretical background is available in the internet[2][3] in this area, but we provide the experimental results after implementing them and comparing them with each other. The significance of our work is that a naive user(who does not understand much high level math) can also distinguish and compare these primality tests.

The paper is organised as follows. Section 2 describes various primality tests that we compare in a brief manner. We have provided easy to follow pseudocode for each of the algorithms. Also we have briefly described the mathematical flavor behind those algorithm in a very concise manner. Section 3 describes the algorithms that generates primes. Section 4 contains the observations and the experiments that we performed after implementing those algorithms. We also highlight our observations in it. Section 6 describes some of the uses of the primes in cryptography.

II. Primality Tests

In this section we discuss various primality tests, both *probabilistic* and *deterministic*.

A. Fermat Primality Test

This test uses *Fermat Little Theorem* to guess if a number is prime or not.

Theorem II.1 (Fermat Little Theorem)**.** *If p is prime, then for any* $1 \leq a \leq n$*, then* $a^{p-1} \cong 1 \mod p$

Algorithm 1 Fermat Primality Testing Algorithm

```
1: procedure FERMAT-PRIMALITY(n, k)   ▷ The number n > 3 to be tested, k is number of times to perform test
2:     if n%2 ≡ 0 then
3:         return composite                                  ▷ The number is composite
4:     while k ≥ 0 do                                        ▷ We check the condition k times
5:         Pick a randomly in range [2,n-2]
6:         if a^{n-1} ≠ 1 mod n then
7:             return composite
8:         else
9:             k ← k − 1
10:    return probably prime
```

B. Trial Division Primality Testing

This is a deterministic test which tests the primality of a number by dividing the number with all the numbers in the range $[2, \sqrt[2]{N}]$. This test takes exponential time in terms of the input number, hence is not feasible for large testings.

Algorithm 2 Trial-Division Testing

```
1: procedure TRIAL-DIVISION(n)                          ▷ The number n to be tested
2:     i ← 2
3:     while i ≤ ∛N do
4:         if n%i ≡ 0 then
5:             return composite                          ▷ The number is composite
6:         else
7:             i ← i + 1
8:     return prime
```

C. Miller-Rabin Primality Test

This is a very famous test invented by Miller and Rabin[4]. It uses the lemma on square roots of *unity* in the field Z_p. Also there are no square roots of *unity* $\bmod p$ other than 1 and -1.

Theorem II.2. *If x, p are positive integers such that $x^2 \equiv 1 \bmod p$ but $x \neq \pm 1 \bmod p$, then p is composite.*

Proof. The proof can be easily seen by using the fact that the number of zeros of a polynomial can not be greater than the degree of the polynomial. □

Theorem II.3. *Let p be prime. We can write $p = 2^s.d$ such that $s, d \geq 0$ and d is odd.Then for each a in Z_p either,*
$a^d \equiv 1 \bmod p$ *or,*
$a^{2^r.d} \equiv -1 \bmod p$ *for some $r \leq s - 1$*

We use the contrapositive of the above theorem to test the primality of number p.

Algorithm 3 Miller-Rabin Primality Test

```
1: procedure MILLER-RABIN(n, k)        ▷ n is the number whose primality is to be tested and k is the number of trials
2:     if n%2 ≡ 0 then
3:         return composite
4:         i ← 0
5:         s ← 0
6:         d ← n − 1
7:         while d%2 ≡ 0 do
8:             s ← s + 1
9:             d ← d/2
10:        while i ≤ k do
11:            pick a random a in range[2, n − 1]
12:            x ← a^d mod n
13:            if x ≠ 1 and x ≠ n − 1 then
14:                return composite
15:            for j = 1, j ≤ s − 1 do
16:                x ← x^2 mod n
17:                if x ≡ 1 mod n then
18:                    return composite
19:                if x ≡ −1 mod n then
20:                    break
21:            if x ≠ −1 mod n then
22:                return composite
23:        return probably prime
24:
```

D. Solovay-Strassen Primality test

This test uses the theorem of primes on Leagendre Symbol[5].

Theorem II.4. *For any prime number p and any integer a,*

$$a^{(p-1)/2} \equiv (a/p) mod p \tag{1}$$

where (a/p) is the legendre symbol such that,

$$(a/p) = \{ 1 \ ,$$

if a is a quadratic residue $\bmod p$ *and* $a \neq 0 \bmod p$
-1, if a is a quadratic non-residue $\bmod p$
0, if $a \equiv 0 \bmod p$

Given a prime p, we can check if this condition holds for all a relatively prime to p. If yes, then we can conclude that its a prime.

Algorithm 4 Solovay-Strassen Primality test

```
procedure SOLOVAY-STRASSEN(n, k)
    if n%2 ≡ 0 then
        return composite
    while k ≥ 0 do
        Choose a random number in range[2, n − 1]
        x ← (a/p)
        if x ≡ 0 or a^((n−1)/2) ≠ x mod n then
            return composite
        k ← k − 1
    return probably-prime
=0
```

E. Lucas-Lehmer Primality test

This is a primality test for Mersenne numbers[**?**] i.e the numbers which are one less than power of 2. These numbers have important uses in Cryptography.

$$M_n = 2^n - 1 \tag{2}$$

If n is composite(n = a.b), then M_n has $2^a - 1$ and $2^b - 1$ as factors. Hence, n is assumed to be prime. The primality of n can be tested by any method such as trial division as n is exponentially smaller than M_n.
To test the primality, we define

$$s_i = \{ 4 \ , if i = 0 s_(i-1)^2 - 2 \tag{3}$$

Theorem II.5. *M_p is prime if and only if $s_{p-2} \equiv 0 \bmod M_p$*

Algorithm 5 Lucas-Lehmer Primality Test

```
procedure LUCAS-LEHMER(p)
    s ← 4
    M_p ← 2^p − 1
    for i ← 2,i − 2,i ← i + 1 do
        s ← (s^2 − 2) mod M_p
    if s == 0 then
        return prime
    else
        return composite
```

F. AKS primality test

This was the first deterministic primality testing algorithm[6] which ran in polynomial time in terms of number of bits. This test proves that $PRIMES \in P$. This test uses the theorem,

Theorem II.6. *An integer n is prime if and only if the polynomial relation*

$$(x+a)^n \equiv (x^n + a) \bmod n \tag{4}$$

hold for some a coprime to n.

Proof. The proof of it can be easily seen by expanding the LHS and seeing that $ni \equiv 0 mod n$ for all $0 \leq k \leq n$ if and only if n is prime. □

However directly using this would take exponential time as it requires expansion of n terms. However this theorem is an equality in a *polynomial ring* $Z_n[x]$. Hence the previous expression is evaluated for a quotient ring of degree r. Hence we evaluate

$$(x+a)^n \equiv (x^n + a) \bmod (x^r - 1, n) \tag{5}$$

The degree of r can be proved to be polynomial in $log(n)$ and hence this can be tested in polynomial time.

Algorithm 6 AKS primality test

procedure AKS PRIMALITY(n)
  if $n = a^b$ for $a \in N$ and $b \geq 1$ **then**
    return composite
  Find smallest r s.t $O_r(n) \geq log^2(n)$ ▷ $O_r(n)$ is the multiplicative order of $n \bmod r$
  For all $a \leq r$ check if a is coprime to n.
  if $1 \leq gcd(a, n) \leq n$ for some $a \leq r$ **then**
    return composite
  if $n \leq r$ **then**
    return prime
  for $a = 1$ to $\sqrt[2]{\phi}(r)log(n)$ **do** ▷ $\phi(r)$ is the euler totient function of r
    if $(x+a)^n \neq (x^n + a) \bmod (x^r - 1, n)$ **then**
      return composite
  return prime

Although this algorithm is deterministic, its takes much more time in comparison to the probabilistic tests and hence is not much used for practical purposes.

G. Pocklington-Lehmer primality Test

This test relies on *Pocklington criterion*[7] which is,

Theorem II.7. *For a number n, let there exist a and q such that*

- *q is prime and q divides $n-1$ and $q \geq \sqrt[2]{n} - 1$*
- *$a^{n-1} \equiv 1 mod n$*
- *$gcd(a^{\frac{n-1}{q}} - 1, n) = 1$*

Then n is prime.

Algorithm 7 Pocklington-Lehmer Test

procedure POCKLINGTON-LEHMER(n, k)
  while $k \geq 0$ **do**
    Choose a random prime q in set [3,n-1]
    for $a2$ upto a **do**
      if $n - 1 \% q 0$ and $a^{n-1} \equiv 1 mod n$ and $gcd(a^{\frac{n-1}{q}} - 1, n) = 1$ **then**
        return prime
    $k \leftarrow k - 1$
  return probably composite

H. Other tests

Apart from the above mentioned tests, many other ways of primality testing exists. Recently $elliptic curve$ properties have been widely used in testing the primality of a number. These tests are fast but are more complicated in nature.

III. Primes Generation

Large prime numbers are required for the security of encryption systems such as RSA and also in random number generators. The traditional method of generating primes has been using *sieves*. We discuss two such tests here.

A. Sieve of Eratosthenes

This is an ancient method of generating primes sequentially. The basic idea for generating all primes upto number n is to keep excluding those numbers from list which are divided by any number less than it in a sequential manner.

Algorithm 8 Sieve of Eratosthenes

procedure SIEVE-ERATOSTHENES(n) ▷ Generating Primes Upto number n
  Maintain an array A of length n denoting if A[i] is prime. Initialise all to true
  for $i \leftarrow 2$ upto $\sqrt[2]{n}$ **do**
    if $A[i] ==$ True **then**
      Make all the values of A[2*i], A[3*i], A[4*i] upto A[k*i] s.t $k * [i] \leq n$ false
  return numbers s.t A[i] is true

B. Sieve of Atkin

This is a recent algorithm [8]. This algorithm uses the concept of modulo 60 division and the properties of the solution of the elliptic curves based on these modulo numbers. This is not an intuitive as 8, but its asymptotic time complexity is faster than Eratosthenes. This test involves more computations as compared to 8, hence although asymptotically fast, it may run slower for many cases as compared to Eratosthenes.

Algorithm 9 Sieve of Atkin

procedure SIEVE OF ATKIN(n) ▷ Find all primes upto n
  Maintain an Array A of n boolean initialised to $True$
  $r \leftarrow \sqrt[2]{n}$
  for $i \leftarrow 1$ upto r **do**
    for $j \leftarrow 1$ upto r **do**
      $l \leftarrow 4 * i^2 + j^2$ ▷ Checking for the first elliptic curve
      if $l \leq n$ and ($n\%$ 12 == 1 or n%12 == 5) **then**
        Change state of $A[l]$
      $l \leftarrow 3 * i^2 + j^2$ ▷ Checking for the second elliptic curve
      if $l \leq n$ and $n\%$ 12 == 7 **then**
        Change state of $A[l]$
      $l \leftarrow 3 * i^2 - j^2$ ▷ Checking for the third elliptic curve
      if $l \leq n$ and ($i \geq j$ and n%12 == 11) **then**
        Change state of $A[l]$
  for $i \leftarrow 5$ upto n **do**
    if A[i] == $True$ **then**
      $k \leftarrow i^2$
      for $j \leftarrow 1$ upto k **do**
        Flip A[$i * j$]
  return A[i] $1 \leq i \leq n$ s.t A[i] == True

IV. Comparison and Performance

A. Specification

We compare the above algorithms by implementing them in *Python* and using the GMP library in *C++* for AKS test. These algorithms were run in an 8-GB RAM machine in an Intel Core 2.10GHz processor.

B. Experimental Observations

We present observations as part of separate tables for both Primality testing and Prime Number Generation.[1]

MAX_PRIME = 15485863			
Algorithm Name	Time Taken (in sec)	Number of times(in Probabilitsitc Cases)	False Positives
Trial Division	140.81185506	-	All correct(1000000)
Fermat's Test	39.9157860279	10	39
	22.7095820904	1	1387
	342.412580967	100	2
Miller Rabin Test	52.5704090595	10	0
	31.9195740223	1	276
	461.214740038	100	0
Solovay-Strassen Test	158.615675926	10	0
	81.6812670231	1	468
	2684.52206802	100	0
AKS test	32936.21855	-	All correct

MAX_PRIME = 35485863			
Algorithm Name	Time Taken (in sec)	Number of times(in Probabilitsitc Cases)	False Positives
Trial Division	432.937591791	-	All correct(2174707)
Fermat's Test	89.0880458355	10	48
	52.9203732014	1	1948
	484.854558945	100	6
Miller Rabin Test	118.651042938	10	0
	73.3548989296	1	367
	748.557681084	100	0
Solovay-Strassen Test	378.513648987	10	0
	221.4035079481	1	712
	2282.13815999	100	0
AKS test	50773.316124	-	All correct

1) Primality Testing Algorithms:

MAX_PRIME = 15485863		
Algorithm Name	Time Taken	Limit of Generation
Sieve of Eratosthenes	16.8654940128	15485863
	0.582002162933	1000000
	32.0918030739	35485863
Sieve of Atkin	53.2122488022	15485863
	1.92054605484	1000000
	69.412184	35485863

Time and Space Complexity of Algorithms		
Algorithm Name	Time Complexity	Space Complexity
Sieve of Eratosthenes	$O(nloglogn)$	$O(n)$
Sieve of Atkin	$O(n)$	$O(n)$
Trial Division	$O(\sqrt[2]{n})$	$O(1)$
Fermat's test	$O(k.log^2n.loglogn.logloglogn)$	$O(1)$
Rabin Miller Test	$O(k.log^3n)$	$O(1)$
Solovay Strassen Test	$O(k.log^3n)$	$O(1)$
AKS test	$O(log^6n)$	$O(1)$
Lucas Lehmer Test	$O(p^3)$ for M_p	O(1)

2) Primality Generation Algorithms: Here in space complexity, I have assumed modular exponentiation can be calculated using constant space. If not, then we can perform it in $O(logn)$ space for each algorithm.

3) Observations:

- Although the time complexity of Sieve of Atkin is better than Sieve of Eratosthenes in generating primes, in practive 8 performs better. It is due to large computational complexity require for 9.
- For getting very large primes, these methods take a very large amount of memory. For getting prime upto 373587883, we were not able to perform this experiment as the RAM was not sufficient. Hence a parallel or distributed version should be used when calculating primes for very large values.

[1]The codes for these algorithms can be found at https://github.com/mssharma5523/Primality-Testing-Comparison

- Probabilistic tests are faster than Deterministic tests and for testing large primes this time difference between them increases in multiples. Also Fermat's test is faster than Miller rabin and Solovay-Strassen in practice also due to very less computation required.
- One of the benchmarks for comparing the probabilistic tests are the number of False positives detected by these tests. We can see that Fermat has the most *false positives* whereas Miller Rabin Algorithm has the least. Also Miller Rabin performs better than Solovay-Strassen in terms of time. Hence Miller Rabin is the most preferred primality test used.
- Also as the number of iterations k increases, the false positives decrease. The false positive will further we removed if we test with prime numbers as the random numbers. Also increasing the number of primes does not affect the number of false positives in Miller Rabin and Solovay Strassen test(with large k, say ≥ 10). Hence these tests can be safely used in equivalence to the deterministic tests for large primes also.
- The Lucas Lehmer test for generating Mersene primes was performed upto the 27^{th} Mersene prime, i.e M_{44497}. This took about 72536 sec to calculate the primes upto this.
- The Pocklington test, although designed for normal primes, takes very large amount of time due to its high computational complexity. We could verify only upto the number 8329 for primality and it took about 50000 seconds to verify.

V. Uses in Cryptography

Prime Numbers form the backbone of *Public-key* cryptography. The difficulty in factoring a number into its prime factors is utilised in the *RSA* encryption algorithm. Also very large primes are used as private keys in the algorithm. Also their very special properties, such as that of Integer Field have large scale uses in other areas of cryptography. Also large uses of primes take place in the field of information theory. Also they are also used in generating pseudo random numbers, such as in *Lehmer Random Number Generator*.

VI. Conclusion

We conclude the performances of different primality tests. We can see that deterministic tests take longer time as compared to the probabilistic tests. Also, by increasing the number of iterations, k, we can decrease the probability of False positives to occur. Also, The test for generating primes upto very large numbers are computationally expensive (as seen for Sieve of Eratostheneses). Hence, it is better to check a random number to be a prime or not than to generate the prime upto those numbers.

Acknowledgment

I would like to thank Dr. S Nandi for giving me the opportunity to work in this project and also to our TA for this course Mayank Agrawal, for helping me in how to go on in this Project.

References

[1] G. L. Miller, "Riemann's hypothesis and tests for primality," *Journal of Computer and System Sciences*, vol. 13, 1976.
[2] C. Pomerance, "Recent developments in primality testing," *The Mathematical Intelligencer*, vol. 3, no. 3, pp. 97–105, 2009. [Online]. Available: http://dx.doi.org/10.1007/BF03022861
[3] S. LANCE, "A survey of primality tests," 2014.
[4] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, 1980.
[5] R. Solovay and V. Strassen, "Erratum: A fast monte-carlo test for primality," *SIAM J. Comput*, 1977.
[6] M. Agrawal, N. Kayal, and N. Saxena, "Primes is in p," *Ann. of Math*, vol. 2, pp. 781–793, 2002.
[7] D. H. Lehmer, "Tests for primality by the converse of fermat's theorem," *Bull. Amer. Math. Soc.*, pp. 327–340, 1927.
[8] A. O. L. ATKIN and D. J. BERNSTEIN, "Prime sieves using binary quadratic forms," *MATHEMATICS OF COMPUTATION*, vol. 73, 2003.