

19uec019

by Yagyik Prajapat

Submission date: 29-Nov-2022 02:10PM (UTC+0530)

Submission ID: 1966101605

File name: BTP_REPORT-2.pdf (2.16M)

Word count: 8808

Character count: 40961

Factorization into primes and their applications

32

Project report submitted in partial fulfillment
of the requirements for the degree of

Bachelor of Technology

in

Electronics and Communication Engineering

by

19uec019 - Yagyik D. Prajapat

19uec023 - Hitesh Goyal

19ucc023 - Mohit Akhouri

Under Guidance of

Dr. Neeraj



5

Department of Electronics and Communication Engineering
The LNM Institute of Information Technology, Jaipur

November 2022

The LNM Institute of Information Technology
Jaipur, India

CERTIFICATE

This is to certify that the project entitled "Factorization into primes and their applications" , submitted by Yagyik D. Prajapat (19uec019), Hitesh Goyal (19uec023) and Mohit Akhouri (19ucc023) in partial fulfillment of the requirement of degree in Bachelor of Technology (B. Tech), is a bonafide record of work carried out by them at the Department of Electronics and Communication Engineering, The LNM Institute of Information Technology, Jaipur, (Rajasthan) India, during the academic session 2021-2022 under my supervision and guidance and the same has not been submitted elsewhere for award of any other degree. In my/our opinion, this report is of standard required for the award of the degree of Bachelor of Technology (B. Tech).

Date

Adviser: Dr. Neeraj

Acknowledgments

We would like to express our sincere and heartfelt gratitude to our mentor and supervisor, Dr. Neeraj for giving us this opportunity to work on this project under his guidance and whose expertise in this field, inspiring ideas and his understanding, patience and mentorship gave us a sense of direction and was very instrumental to complete the project in an effective and sagacious way.

This project helped us in enriching our knowledge about the different concepts and techniques used in the project and how to use them in correct way. The project helped us in expanding the spectrum of our knowledge further more. We would also like to thank our family and friends who helped in keeping ourselves motivated throughout this journey.

Abstract

4

The project focus on the primality tests and factorization algorithms. Testing whether a number is prime or not and finding their prime factors is one of the most fundamental problem in computational number theory. Primality tests and factorization algorithms have got the interests of humans since ancient times. Primality tests and factorization algorithms have wide applications in computer science particularly in the field of cryptography.

In the first part of the project, We will discuss the well known algorithms and also address the attempts to develop a reliable and efficient method for testing primality. A primality test is basically a function which determines if a given integer greater than 1 is prime or composite. The primality tests can be classified as either probabilistic primality tests, deterministic primality tests or non-deterministic primality tests. A probabilistic primality test is a nondeterministic primality test which returns whether the input integer n is not a prime or is prime to some given degree of likelihood. A deterministic primality test returns whether the input integer n is prime or composite. A non-deterministic primality test returns whether the input integer n is not prime or it may be a prime. Some of the popular primality tests include Sieve of Erastosthenes, Fermat Test, Lucas Test, Miller-Rabin primality test and many more. So the point comes where primality tests are used, everytime someone uses the RSA public key cryptosystem and they need to generate a private key consisting of two large prime numbers and a public key consisting of their product. In this case, primality tests come handy in checking rapidly if a number is prime or not. We have discussed the algorithms of primality test and their implementation in fortran. We have also done the complexity analysis of the algorithms.

The second part of the project focuses on the factorization algorithms. We have explored the methods to generate all prime factors of any given integer greater than 1. Determining prime factorization of a given integer has been an active area of mathematical research for over 2300 years. Some of the well known integer factorization algorithms include Pollard rho method, Pollard p-1 method and fermat factor base method. We have discussed the algorithms of integer factorization methods and also implemented their code. We have also done the complexity analysis of these factorization algorithms.

Contents

1	Primality Tests	1
1.1	Sieve of Erastosthenes	1
1.1.1	Introduction	1
1.1.2	Algorithm	2
1.1.3	Code	3
1.1.4	Results	4
1.1.5	Complexity Analysis	5
1.2	KS Primality Test	6
1.2.1	Introduction	6
1.2.2	Algorithm	7
1.2.3	Code	8
1.2.4	Results	9
1.2.5	Complexity Analysis	10
1.3	Tri-Division	11
1.3.1	Introduction	11
1.3.2	Algorithm	12
1.3.3	Code	13
1.3.4	Results	14
1.3.5	Complexity Analysis	15
1.4	Wilson's Primality Test	16
1.4.1	Introduction	16
1.4.2	Algorithm	17
1.4.3	Code	18
1.4.4	Results	19
1.4.5	Complexity Analysis	20
1.5	Lucas Lehmer Primality Test	21
1.5.1	Introduction	21
1.5.2	Algorithm	22
1.5.3	Code	23
1.5.4	Results	24
1.5.5	Complexity Analysis	25
1.6	Fermat Primality Test	26
1.6.1	Introduction	26
1.6.2	Algorithm	27
1.6.3	Code	28
1.6.4	Results	31
1.6.5	Complexity Analysis	32

1.7	Miller Rabin Primality Test	33
1.7.1	Introduction	33
1.7.2	Algorithm	34
1.7.3	Code	35
1.7.4	Results	37
1.7.5	Complexity Analysis	38
1.8	Lucas Primality Test	39
1.8.1	Introduction	39
1.8.2	Algorithm	40
1.8.3	Code	41
1.8.4	Results	43
1.8.5	Complexity Analysis	44
2	Prime Factorization Algorithms	45
2.1	Fermat Factorization	45
2.1.1	Introduction	45
2.1.2	Algorithm	46
2.1.3	Code	47
2.1.4	Results	48
2.1.5	Complexity Analysis	49
2.2	Pollard Rho Factorization	50
2.2.1	Introduction	50
2.2.2	Algorithm	51
2.2.3	Code	52
2.2.4	Results	54
2.2.5	Complexity Analysis	55
2.3	Pollard p-1 Factorization	56
2.3.1	Introduction	56
2.3.2	Algorithm	57
2.3.3	Code	58
2.3.4	Results	60
2.3.5	Complexity Analysis	61
3	Conclusion	62
	Bibliography	62

Chapter 1

Primality Tests

This part focuses on the different primality tests both deterministic and non-deterministic. We will first discussed the algorithm of primality tests and then implemented the algorithms in FORTRAN. Later, we have done the complexity analysis of the codes. The different primality tests are discussed as follows.

1.1 Sieve of Erastosthenes

1.1.1 Introduction

³¹ The Sieve of ¹⁶ Erastosthenes is an ancient algorithm for finding the prime numbers upto any given limit. The sieve of erastosthenes is one of the effiecient ways to find all primes smaller than a given input integer n. The idea behind the algorithm is simple, first it creates a table of numbers from 2 to given integer n. Then it takes the prime numbers ⁶⁵ one at a time and remove their multiples from the table. It continues the process until $p \leq \sqrt{n}$ where p is a prime number.

⁴³ The earliest known reference ²³ to the sieve of erastosthenes is in the Nicomachus of Gerasa's *Introduction to Arithmetic*, an early 2nd century CE book. The algorithm of sieve of erastosthenes creates a list of consecutive integers from 2 to n. We find first the smallest prime number which is equal to 2. We then start the algorithm by enumerating through the multiples of p and mark them in the list. Now we will find the smallest number in the list which is greater than p and is not marked. We will follow the same procedure ⁶⁷ for the number and algorithm continues. At last, we will have some unmarked numbers in the list which are the required prime numbers. The algorithm and code for the Sieve of Erastosthenes is discussed in the next section.

1.1.2 Algorithm

Algorithm 1 Sieve of Erastosthenes

```
1: procedure SIEVE(n)                                ▷ Input is n ∈ N
2:   System Initialization
3:   read the value of n
4:   a[1...n] integer array
5:   while j=1 to n do
6:     a[j] ← j
7:   end while
8:   i ← 2
9:   while i2 < n do
10:    if a[i] ≠ 0 then
11:      t ← 2.i
12:      while t ≤ n do
13:        a[i] ← 0
14:        t ← t + i
15:      end while
16:    end if
17:  end while
18:  i ← i + 1
19:  while j = 2 to n do
20:    if a[j] ≠ 0 then
21:      return a[j] is prime
22:    end if
23:  end while
24: end procedure
```

1.1.3 Code

```
1 PROGRAM SieveOfErastosthenes
2
3 ! This is a initial List
4 INTEGER Candidates(999);
5 ! These are Loop variables
6 INTEGER i,j;
7 ! This Loop will initialize the Candidates array
8 DO 10 i=1 , 999
9     Candidates(i) = 1
10 CONTINUE
11
12 Candidates(1) = 0
13
14 ! The below Loop is the main Loop for sieve of erastosthenes
15 ! It finds the multiples of numbers and then delete them from the list
16 i = 1
17 DO WHILE (i .LT. 1000)
18
19     DO WHILE (i .LT. 1000 .AND. Candidates(i) .EQ. 0)
20         i = i+1
21     END DO
22
23     IF (i .LT. 1000) THEN
24         j = 2
25         DO WHILE (j*i .LT. 1000)
26             Candidates(j*i) = 0
27             j = j + 1
28         END DO
29         i = i+1;
30     ENDIF
31 END DO
32
33 ! Below Loop prints the unmarked numbers in the List which are left at the last
34 DO 20 i=1 , 999
35     IF (Candidates(i) .NE. 0) THEN
36         PRINT *,i," is prime";
37     ENDIF
38 20 CONTINUE
39 END
40
```

FIGURE 1.1: FORTRAN Code of Sieve of Erastosthenes

1.1.4 Results

```
2  is prime
3  is prime
5  is prime
7  is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
101 is prime
103 is prime
107 is prime
109 is prime
113 is prime
127 is prime
131 is prime
137 is prime
139 is prime
149 is prime
151 is prime
```

FIGURE 1.2: Results Obtained for the Code

1.1.5 Complexity Analysis

The time taken by the Sieve of Erastosthenes is discussed as below :

- 4 1. It is assumed that the time taken to mark a number as composite is constant, hence the number of times the loop runs is equal to -

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \dots + p \quad (1.1)$$

- 4 2. On taking n common from the above equation , the equation can be rewritten as :

$$n * \left(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \dots + \infty \right) \quad (1.2)$$

- 4 3. The harmonic progression of the sum of primes is as follows :

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \dots = \log(\log(n)) \quad (1.3)$$

- 75 4. If we substitute the above sum in the equation 2, the **time complexity** comes out to be :

$$\mathbf{O(n * \log(\log(n)))} \quad (1.4)$$

- 16 5. The **space complexity** of Sieve of Erastosthenes would be $\mathbf{O(N)}$ where N would be the size of the candidates array in the code.

1.2 AKS Primality Test

1.2.1 Introduction

⁹ The **AKS primality test** (Agrawal–Kayal–Saxena primality test) and it is deterministically correct for any general number.

Features of AKS primality test : The AKS algorithm can be used to verify the primality of any general number given. The maximum running time of the algorithm can be expressed as a polynomial over the number of digits in the target number. The algorithm is guaranteed to distinguish deterministically whether the target number is prime or composite. The correctness of AKS is not conditional on any subsidiary unproven hypothesis. In contrast, Miller's version of the Miller–Rabin test is fully deterministic and runs in polynomial time over all inputs, but its correctness depends on the truth of the yet-unproven generalized Riemann hypothesis. The AKS primality test is based upon the following theorem: An integer n greater than 2 is prime if and only if the polynomial congruence relation :

$$x + a^n \cong (x^n + a)(mod n) \quad (1.5)$$

⁴ holds for some a coprime to n .

Here x is just a formal symbol .The AKS test evaluates the equality by making complexity dependent on the size of r . This is expressed as:

$$x + a^n \cong (x^n + a)mod(x^r - 1, n) \quad (1.6)$$

⁴⁹ which can be expressed in simpler term as :

$$x + a^n - (x^n + a) = (x^r - 1)g + nf \quad (1.7)$$

⁴ for some polynomials f and g .

This congruence can be checked in polynomial time when r is polynomial to the digits of n . The AKS algorithm evaluates this congruence for a large set of a values, whose size is polynomial to the digits of n . The proof of validity of the AKS algorithm shows that one can find r and a set of a values with the above properties such that if the congruences hold then n is a power of a prime.

1.2.2 Algorithm

Algorithm 2 AKS Primality Test

```
1: procedure AKS( $n$ ) ▷ Input is  $n \in \mathbb{N}$ 
2:   System Initialization
3:   Read the value of  $n$ 
4:   If  $\exists a, b > 1 \in \mathbb{N}$  such that  $n = a^b$ , then output composite
5:   Find the minimal  $r \in \mathbb{N}$  such that  $o_r(n) > \log^2(n)$ 
6:   while  $a=1$  to  $r$  do
7:     if  $1 < (a, n) < n$  then
8:       output Composite
9:     end if
10:    end while
11:    if  $r = n$  then
12:      output Prime
13:    end if
14:    while  $a = 1$  to  $\lfloor \sqrt{\phi(r)} \cdot \log(n) \rfloor$  do
15:      if  $(X + a)^n \neq X^n + a \pmod{X^r - 1, n}$  then
16:        output Composite
17:      end if
18:    end while
19:    Return Prime
20: end procedure
```

1.2.3 Code

```

1 !AKS PROGRAM
2 program AKS
3
4 !initializing n as integer and c as integer array
5 integer(kind=16) :: n,c(200)
6
7 read*,n
8
9 !Loop for assigning values of c array to 0
10 do i=1,200
11     c(i) = 0
12 end do
13 ! ending the Loop
14
15 !first element to 1
16 c(1) = 1
17
18 !loop traversing from 1 to n
19 do i=1,n
20     c(1+i) = 1
21
22     do j=i,2,-1
23         c(j) = c(j-1) - c(j)
24     end do
25
26     c(1) = -c(1)
27 end do

```

FIGURE 1.3: Part 1 of FORTRAN Code of AKS Primality Test

```

28
29
30 c(1) = c(1) + 1
31 c(n+1) = c(n+1) - 1
32
33 i = n+1
34 do while(i>0 .and. mod(c(i),n)==0)
35     i=i-1
36 end do
37
38 !if i<1 then the number is prime
39 if (i<1) then
40     print*, " the number is prime"
41 else
42     print*, " the number is not prime" !if i>=1 then the number is not prime
43 end if
44
45 end program AKS

```

FIGURE 1.4: Part 2 of FORTRAN Code of AKS Primality Test

1.2.4 Results

```
113
the number is prime

...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 1.5: Results Obtained for the Code

```
145
the number is not prime

...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 1.6: Results Obtained for the Code

1.2.5 Complexity Analysis

Before going for the complexity analysis of AKS Test, Let us have a look at some theorems and lemma.

Theorem : The AKS algorithm runs in $O(\log^{\frac{21}{2}} n)$ time.

Proof : The step 1 of the algorithm takes $O(\log^3 n)$ time. We can try successive values of r until we find one such that $n^k \neq 1 \pmod r$ for all $k \leq \log^2 n$. For a particular r this would involve at most $O(\log^2 n)$ multiplications modulo r . Since we are multiplying modulo r each product will have factors less than r . Hence, again by our summary, each multiplication takes $O(\log r)$ time. Computing GCD of r numbers where r is bounded by previous step. Computing GCD takes $O(\log n)$ time. Therefore, total time complexity for this step is $O(r \log n) = O(\log^6 n)$. Comparison of r and n can be done by counting the number of digits in n and seeing if r has that many or more. This takes time proportional to the number of binary digits in n , so it takes about $O(\log n)$ time. Next comes the loop which runs for values of a from 1 to $\lfloor \sqrt{\phi(r)} \cdot \log(n) \rfloor$, the time complexity for this step would be $O(\sqrt{r} \log n)$. Finally, the step computes $(X + a)^n$ and $X^n + a \pmod{X^r - 1, n}$. Naively it could take n multiplications to compute $(X + a)^n \pmod{X^r - 1, n}$. This step dominates all the other ones, so the total runtime of our algorithm is indeed $O(\log^{21/2} n)$.

The time complexity of the algorithm may be improved by improving the bounds on r . The best possible scenario would be when $r = O(\log^2 n)$ and in that case we would get a total time complexity of $O(\log^6 n)$.

Lemma : Let $P(m)$ be the greatest prime divisor of m . There exists constants $c > 0$ and n_0 such that, for all $x \geq n_0$:

$$|q| q \in \text{prime}, q \leq x, P(q-1) > q^{2/3} \geq c \cdot \frac{x}{\log x} \quad (1.8)$$

Theorem : The asymptotic time complexity of the AKS algorithm is $O(\log^{15/2} n)$.

Proof : A high density prime q such that $P(q-1) > q^{2/3}$ implies that the algorithm will find a $r = O(\log^3 n)$ with $\phi(r) > \log^2 n$. Using this, the time complexity of the algorithm is brought down to :

$$O(r^{3/2} \log^3 n) = O((\log^3 n)^{3/2} \log^3 n) = O(\log^{15/2} n) \quad (1.9)$$

The overall **Time Complexity** of AKS primality test algorithm is $O(\log^{15/2} n)$. The overall **Space Complexity** of the AKS Primality test algorithm would be $O(N)$ since the space is taken by the intermediate array $C(n)$.

1.3 Trial Division

1.3.1 Introduction

Trial Division is the strenuous but easiest primality test. The essential idea behind trial division test is to see if integer n can be divided by each number less than n . Trial Division was first described by Fibonacci in his book Liber Abaci. The method followed by Trial Division is: Given an integer n , the trial division systematically checks whether any smaller number than n divides n . With ordering, there is no point in testing divisibility by 4 if it already is not divisible by 2. Therefore, major effort can be reduced if we select only prime numbers as candidate factors.

Furthermore, the factors in trial division does not go further than \sqrt{n} because if n is divisible by some number p , then n is equal to product of p and q . When looking for large prime, we could never divide by all primes less than the square root of that number. But we can still use the trial division for pre-screening, that is if we want to know that n is prime, then we divide it by a small million primes, then we apply a primality test.

One of the observations of Trial Division is that it will work if the maximum factor for any number N is always less than or equal to square root of N . The approach for the algorithm of Trial Division is simple, instead of checking factors for N till $N-1$, we check till only \sqrt{N} . The algorithm, code and complexity analysis is discussed in the further sections.

1.3.2 Algorithm

Algorithm 3 Trial Division

```
1: procedure TRIALDIVISION( $n$ ) ▷ Input is  $n \in \mathbb{N}$ 
2:   System Initialization
3:   Read the value of  $n$ 
4:   while  $k=2,3,4,\dots,\lfloor\sqrt{n}\rfloor$  do
5:     if  $n \equiv 0 \pmod{k}$  then
6:       output Composite
7:     end if
8:   end while
9:   Return  $n$  is Prime
10: end procedure
```

1.3.3 Code

```
1 program TrialDivision
2
3 !here kind represent that integer can hold a maximum value of 2^16.
4 integer(kind=16) :: n,i,k,flag=1
5
6 print*, "Enter Number to check for primality"
7 read*, n
8
9 i = 2
10 k = n**(0.5)
11 ! k stores the value of square root of n.
12
13 loop: do while(i<=k)
14     ! Loop will run maximum of square root n times.
15
16     if(mod(n,i) == 0) then
17         ! If we have find one factor, we set flag as 1 and exit from loop.
18         flag=0
19         exit loop
20     endif
21
22     i = i+1
23 end do loop
24
25 if(flag==1) then
26     ! If flag is set as 1, means number is prime.
27     print*, "The number is prime"
28 else
29     ! If flag is set as 0, means number is non prime.
30     print*, "The number is not prime"
31 end if
32
33 end program TrialDivision
```

FIGURE 1.7: FORTRAN Code of Trial Division

1.3.4 Results

```
Enter Number to check for primality
17
The number is prime

...Program finished with exit code 0
Press ENTER to exit console. □
```

FIGURE 1.8: Results Obtained for the Code

```
Enter Number to check for primality
16
The number is not prime

...Program finished with exit code 0
Press ENTER to exit console.
```

FIGURE 1.9: Results Obtained for the Code

1.3.5 Complexity Analysis

The **time complexity** analysis of the Trial Division is as follows. The trial division algorithm to check if number N is prime or not works by trying to divide N by all integers in the range $2, 3, \dots, \lfloor \sqrt{n} \rfloor$. In the worst case, Trial division requires $O(\sqrt{N})$ divisions to be made.

Trial Division algorithm runs in the time $O(\sqrt{n} \cdot \log^2(n))$, but this running time is exponential in the input size since the input represents n as a binary number with $\lfloor \log_2(n) \rfloor$ digits. **Space Complexity** of Trial Division Algorithm would be **O(1)** since we are using only space of variables to implement the Algorithm.

1.4 Wilson's Primality Test

1.4.1 Introduction

Wilson's Primality Test is based on the **Wilson's Theorem**. Wilson's theorem states that a natural number $n > 1$ is a prime number iff we see that product of positive integers less than n is one less than a multiple of n . The Wilson's theorem was stated by Ibn al-Haytham (at around 1000 AD) and in the 18th century by John Wilson. Lagrange gave the first proof for the theorem in 1771. The Wilson's theorem in formal terms can be stated as follows :

Theorem : Let $n \in \mathbb{N}$. Then n is prime if and only if $(n-1)! \equiv -1 \pmod{n}$.

Proof: For the forward part, Suppose n is prime. Then every integer in the interval $[2,3,4,\dots,n-2]$ is coprime to n and has a unique inverse modulo n . Therefore,

$$\prod_{2 \leq j \leq n-2} j \equiv 1 \pmod{n} \quad (1.10)$$

and we know that $(n-1) \equiv -1 \pmod{n}$. Hence,

$$\prod_{2 \leq j \leq n-1} j \stackrel{1}{=} (n-1)! \equiv -1 \pmod{n} \quad (1.11)$$

Now let us look at the converse part, suppose that n is composite. Then $1,2,3,\dots,n-1$ contains all prime factors of n , which implies that $(n-1)! \not\equiv -1 \pmod{n}$ (because if $(n-1)! \equiv -1 \pmod{n}$, then a factor of n , say d will also satisfy this congruence. One can see that $(n-1)! \equiv 0 \pmod{d}$). From the Wilson's characterization of primes, we can determine the primality of an integer n by calculating $(n-1)! \pmod{n}$. But this computation would require $(n-1)$ multiplications, making it very time consuming.

1.4.2 Algorithm

Algorithm 4 Wilson's Primality Test

```
1: procedure WILSON(n)                                ▷ Input is n ∈ N
2:   System Initialization57
3:   Read the value of n
4:   if (n-1)! ≡ -1(mod n) then
5:     Output Prime
6:   end if
7:   otherwise, return Composite
8: end procedure
```

1.4.3 Code

```

1 ! This is a recursive function which implements the Wilson's Theorem
2 recursive function fact(p) result(ans)
3     integer, intent(in) :: p
4
5     if(p<=1) then
6         ans=1
7     else
8         ans=p*fact(p-1)
9     end if
10 end function
11
12 ! This is the function which will take input an integer p
13 ! and it will check whether it is prime or not
14 function isPrime(p) result(ans)
15     integer, intent(in) :: p
16     integer :: ans
17     integer :: p1
18     integer :: fans
19     if(p==4) then
20         ans=0
21     else
22         ! We have used rshift function to shift the bits to right 1 place
23         p1=rshift(p,1)
24         fans = fact(p1)
25         ! Here, mod indicates modulus function of fans with p
26         ans=mod(fans,p)
27     end if
28 end function
29
30 ! Main function starts here, entry of test integer n is done
31 program main
32     integer :: n
33
34     read*,n
35
36     if(isPrime(n)==0) then
37         print*, "The number is not prime"
38     else
39         print*, "The number is prime"
40     end if
41
42
43 end program

```

FIGURE 1.10: FORTRAN Code of Wilson's Primality Test

1.4.4 Results

```
31
The number is prime

...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 1.11: Results Obtained for the Code

```
25
The number is not prime

...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 1.12: Results Obtained for the Code

1.4.5 Complexity Analysis

The Wilson's Primality test makes use of recursive factorial function to implement the Wilson's Theorem and check whether a number is prime or not. The Time and Space Complexity Analysis of Wilson's Theorem can be summarized as follows :

1. Input of integer takes $O(1)$ time.
2. Now isPrime() function is called, in that rshift function is used and mod function is used along with recursive factorial function. rshift function and mod function are inbuilt functions, they will take $O(1)$ time.
3. recursive factorial function will take $O(N)$ time.
4. Hence, Total **Time Complexity** = $O(1) + O(1) + O(N) = O(N)$.
5. Now auxillary space used in recursion would be $O(N)$. Hence **Space Complexity** of Wilson's Primality Test is **$O(N)$** .

1.5 Lucas Lehmer Primality Test

1.5.1 Introduction

2

The **Lucas-Lehmer Primality Test** is a primality test for Mersenne numbers. The test was formulated by Edouard Lucas in 1876 and was subsequently improvised by Derrick Henry J. Lehmer in the 1930s. Now let us talk in brief about Mersenne numbers.

Mersenne numbers is a number which is prime and is one less than a power of two. Mersenne numbers can be written in the form of $M_n = 2^n - 1$ for some integer n. Mersenne numbers are named after Marin Mersenne who was a French Minim friar who studied about Mersenne numbers in the 17th century.

The Lucas-Lehmer test works as follows :

- Let $M_p = 2^p - 1$ be the Mersenne number to test and p is an odd prime.
- The primality of p can be checked with the help of any primality algorithm such as Trial Division.
- Here, p is exponentially smaller than M_p .
- We will define a sequence $\{s_i\}$ for all $i \geq 0$ by :

$$s_i = 4, i = 0 \quad (1.12)$$

$$s_i = s_{i-1}^2, \text{otherwise} \quad (1.13)$$

- 2
- Some of the terms of this sequence are 4, 14, 194, 37634. Now M_p is prime iff :

$$s_{p-2} \equiv 0 \pmod{M_p} \quad (1.14)$$

39 The number $s_{p-2} \pmod{M_p}$ is called the Lucas-Lehmer residue of p. Lucas-Lehmer is a deterministic primality test algorithm and it is the last stage in the procedure which is employed by 53 Great Internet Mersenne Prime Search Distributed computing project to find Mersenne primes.

1.5.2 Algorithm

Algorithm 5 Lucas-Lehmer Primality Test

```
1: procedure LUCAS-LAHMER
2:   System Initialization
3:   Declare variable i64
4:   Declare variables s and n
5:   Declare variable i and exponent
6:   while exponent = 2 to 31 do
7:     if exponent = 2 then
8:       Let s=0
9:     else
10:      Let s=4
11:    end if
12:   end while
13:   Let n = 2.i64**exponent - 1
14:   while i = 1 to exponent-2 do
15:     s = (s*s - 2) % n
16:   end while
17:   if s=0 then
18:     return Prime
19:   end if
20: end procedure
```

1.5.3 Code

```
1 PROGRAM LUCAS_LEHMER
2 IMPLICIT NONE
3
4 ! Declaration of variables and parameters
5 INTEGER, PARAMETER :: i64 = SELECTED_INT_KIND(18)
6 INTEGER(i64) :: s, n
7 ! For storing of exponents from 2 to 31
8 INTEGER :: i, exponent
9
10 ! This is the main loop for calculation of prime numbers for Lucas-Lehmer Test
11 DO exponent = 2, 31
12
13 ! if-else part for setting s=0 or 4 according to value of exponent
14 IF (exponent == 2) THEN
15   s = 0
16 ELSE
17   s = 4
18 END IF
19
20 n = 2_i64**exponent - 1
21
22 ! Check for i from 1 to exponent-2, calculate modulus of s*s-2 and n
23 DO i = 1, exponent-2
24   s = MOD(s*s - 2, n)
25 END DO
26
27 ! if s=0, then print the prime numbers
28 IF (s==0) WRITE(*,"(A,I0,A)") "M", exponent, " is PRIME"
29 END DO
30
31 END PROGRAM LUCAS_LEHMER
```

FIGURE 1.13: FORTRAN Code of Lucas-Lehmer Primality Test

1.5.4 Results

```
M2 is PRIME
M3 is PRIME
M5 is PRIME
M7 is PRIME
M13 is PRIME
M17 is PRIME
M19 is PRIME
M31 is PRIME

...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 1.14: Results Obtained for the Code

1.5.5 Complexity Analysis

The **time complexity** analysis of Lucas-Lehmer Primality Test can be summarized as follows. The algorithm of Lucas-Lehmer Primality Test consists of two expensive operations during each iteration which is multiplication of s with itself $s \times s$, and mod M operation. The mod M operation can be made efficient if we observe that :

$$k \equiv (k(\text{mod } 2^n)) + \lfloor \frac{k}{2^n} \rfloor (\text{mod } 2^n - 1) \quad (1.15)$$

The above equation says that the least significant n bits of k plus the remaining bits of k are equivalent to k modulo $2^n - 1$. The equivalence can be used repeatedly until atmost n bits will remain. The remainder after dividing k by the Mersenne number $2^n - 1$ is computed using division process. Now $s \times s$ will never exceed M^2 and this converges in at most 1 p-bit addition and it can be done in linear time. Now asymptotic time complexity of Lucas-Lehmer Test depends only on multiplication algorithm. The multiplication will require $O(p^2)$ bit-level operations to square a p-bit number.

Lucas-Lehmer Primality Test when used with Fast fourier tranform has a complexity of $O(\log^2 N \log \log N)$ where N represents the Mersenne prime number. The complexity can be further reduced down to $O(n^2 \log n)$. The **space complexity** of Lucas-Lehmer test is O(1) since no extra or auxillary space is used.

1.6 Fermat Primality Test

1.6.1 Introduction

11

The **Fermat primality test** is a primality test, giving a way to test if a number is a prime number, using Fermat's little theorem and modular exponentiation.

Fermat's Little Theorem states that if a is relatively prime to a prime number p , then $a^{p-1} \equiv 1 \pmod{p}$. Fermat's little theorem is not true for composite numbers generally, and so it is an excellent tool to use to test for the primality of a number. Fermat's little theorem states that if p is prime and a is not divisible by p , then :

$$a^{p-1} \equiv 1 \pmod{p}. \quad (1.16)$$

2

To test a number (let p) whether that is prime, then first pick random integers 'a' which are not divisible by p and observe whether the equality holds. If it is proven that the equality doesn't hold then we can say that p is composite and not prime. This congruence is unlikely to hold for a random a if p is composite. Therefore, if the equality does hold for one or more values of a , then we say that p is probably prime. However, note that the above congruence holds trivially for $a^{p-1} \equiv 1 \pmod{p}$, because the congruence relation is compatible with exponentiation. That is why one usually chooses a random a in the interval $1 < a < p-1$. Any a such that $a^{n-1} \equiv 1 \pmod{n}$ when n is composite is known as a **Fermat liar**. In this case n is called Fermat pseudoprime to base a . If we do pick an a such that $a^{n-1} \not\equiv 1 \pmod{n}$ (! \equiv means not equal) then a is known as a Fermat witness for the compositeness of n . The idea of algorithm is simple :

1. Pick a positive integer $x < n$.
2. Check Whether x is Fermat witness.
3. If so, then output "composite". Otherwise output "probably prime".

Now, to determine whether x is Fermat witness for n , we need to compute $x^{n-1} \pmod{n}$. The obvious way of doing this requires $n-2$ iterations of mod n multiplication. But using the binary expansion of $n-1$ and repeated squaring method, we can reduce this to $O(\log n)$ multiplication operations.

1.6.2 Algorithm

Inputs are n and k, n is a number which we want to know whether it is prime or not and k is factor that determines how many number of times we have to test. Higher the k means probability of our composite result is correct and for prime, our result is always correct.

Output : It shows whether n is composite or prime.

Algorithm 6 Fermat Primality Test

```
1: procedure FERMAT-PRIMALITY( $n$ )                                 $\triangleright$  Input is  $n \in \mathbb{N}$ 
2:   System Initialization
3:   read the value of  $n$ 
4:   Choose  $x \in \{1, 2, 3, \dots, n-1\}$  uniformly at random
5:   if  $x^{n-1} \neq 1 \pmod{n}$  then
6:     return Composite
7:   else
8:     return Probably Prime
9:   end if
10: end procedure
```

1.6.3 Code

```
1 !FERMAT PRIMALITY TEST
2
3 !function prime for checking if a number is prime or not
4 function power(a,n,p) result(res)
5
6     res = 1
7     a = mod(a,p) !storing (a mod p) in a
8
9     ! do while Loop until n>0
10    do while(n > 0)
11
12        !in if when n mod 2 == 1
13        if(mod(n,2) .eq. 1) then
14            res = mod(res*a,p)
15            n = n - 1 ! decreamenting value of n by 1
16        else
17            a = mod((a**2),p)
18            n = n / 2      ! dividing n by 2
19
20        end if
21
22    end do
23    ! ending the Loop
24
25    res= mod(res,p) !storing result in res variable
26
27 end function power
28
```

FIGURE 1.15: Part 1 of FORTRAN Code of Fermat Primality Test

```
29 function isPrime(n,k) result(res)
30 !initializing variables like n, k, res, a ,lr, rr
31     integer, intent (in) :: n,k
32     integer :: res
33     real :: a
34     integer :: lr,rr
35
36
37 if(n.eq.1 .or. n.eq.4) then
38     res = 0
39 else if(n.eq.1 .or. n.eq.3) then
40     res = 1
41 else
42     iloop: do i=1,k    !iterating in Loop for 1 to k
43         !calling this function everytime to get a random number
44         call random_number(a)
45         lr=2
46         rr=n-2
47         a=(rr-lr+1)*a
48         ! a = random_number(2,n-2)
49         if(power(a,n-1,n) .ne. 1) then
50             res = 0
51         end if
52     end do iloop
53
54 end if
55
56 end function isPrime
```

FIGURE 1.16: Part 2 of FORTRAN Code of Fermat Primality Test

```
program main
integer:: k
k=3
if(isPrime(113,k).eq.1) then
    print*, "113 is not prime"
else
    print*, "113 is prime"
end if

if(isPrime(145,k).eq.1) then
    print*, "145 is not prime"
else
    print*, "145 is prime"
end if
end program
```

FIGURE 1.17: Part 3 of FORTRAN Code of Fermat Primality Test

1.6.4 Results

```
113 is prime
145 is not prime

...Program finished with exit code 0
Press ENTER to exit console.□
```

FIGURE 1.18: Results Obtained for the Code

1.6.5 Complexity Analysis

23

The **time complexity** of the Fermat Primality Test is $O(K * \log(N))$. Let us discuss how this time complexity is derived. In `isPrime()` function, a loop is running K times and each of the time power function is called. So, Time complexity of prime function is : $O(K * (\text{Time complexity of power function}))$.

16

The time complexity of Power function can be derived as follows : In power function, if n is even, $n=n/2$ else $n=n-1$ to make sure that n is even.

$$T(n) = T(n/2) + c_1 \quad (1.17)$$

63

The above equation holds if n is even.

$$T(n) = T(n-1) + c_2 \quad (1.18)$$

6

The above equation holds if n is odd. Hence, $T(n) = T((n-1)/2) + c_1 + c_2$. After successively solving the equation of $T(n)$, we get :

- $T(n) = T(n/2) + c$, $c > c_1$
- $T(n) = T(n/4) + 2c$
- $T(n) = T(n/4) + 2c$
-contd
- $T(n) = T(n/(2^t)) + tc$

Now after analysing the above equation we come to the conclusion that : $n/(2^t) = 1$ reduces to $2^t = n$ reduces to $t = \log(n)$. Hence, power function takes $\log(n)$ time. Therefore, overall time complexity of Fermat Primality Test comes out to be $O(K * \log(N))$. The **space complexity** of Fermat Primality Test comes out to be $O(1)$ since there is no need for extra space, so the space complexity is constant.

68

1.7 Miller Rabin Primality Test

1.7.1 Introduction

34

The **Miller Rabin Primality Test** is a probabilistic primality test which means it will return whether a number is not prime or it is prime to some given degree of likelihood. Miller Rabin Primality Test was discovered by Gary L. Miller in 1976. Initially, Miller's version of the Primality Test was deterministic but correctness of it depends on the unproven extended Riemann hypothesis. The algorithm of Miller Rabin was modified by Michael O. Rabin in 1980 to obtain an unconditional probabilistic algorithm.

31

Miller Rabin test is of much historical significance since it is used in the search for a polynomial-time deterministic primality test. It is one of the simplest and fastest primality test ever known. The mathematical concepts used in Miller Rabin primality test are : Strong probable primes which are prime numbers to base a if congruence relation $a^d \equiv 1 \pmod{n}$ holds. Choices of bases for Miller Rabin test is also very important. A naive solution is to try all possible bases which can yield an inefficient deterministic algorithm. Another possible solution can be to pick the base at random which will yield a faster probabilistic test. The Miller-Rabin test is an advanced version of Fermat Primality Test. The Miller-Rabin test is based on the Fermat's Little Theorem and the following lemma :

Lemma (Fake Square root lemma) : If x, n are positive integers such that $x^2 \equiv 1 \pmod{n}$ but $x \neq \pm 1 \pmod{n}$, then n is composite.

Proof : From the hypothesis of lemma, n divides $x^2 - 1 = (x-1)(x+1)$, but n divides neither $x+1$ nor $x-1$. This is impossible when n is prime, hence n is composite.

12

Idea behind Miller-Rabin Test : The idea of the test is to pick a random number x in $\{1, 2, \dots, n-1\}$ and use it to try finding either a Fermat witness or a fake square root of $1 \pmod{n}$.

1.7.2 Algorithm

Algorithm 7 Miller-Rabin

```
1: procedure MILLER-RABIN( $n$ ) ▷ Input is  $n \in \mathbb{N}$ 
2:   System Initialization
3:   read the value of  $n$ 
4:   if  $n > 2$  and  $n$  is even then
5:     return Composite
6:   end if
7:   Choose  $x \in \{1, 2, 3, \dots, n-1\}$  uniformly random.
8:   if  $x^{n-1} \neq 1 \pmod{n}$  then
9:     return Composite
10:   else
11:     Factor  $n-1 = 2^s \cdot t$ , where  $t$  is odd
12:     Compute  $u_i = x^{2^i \cdot t} \pmod{n}$ , where  $0 \leq i < s$ .
13:     If there is an  $i$  such that  $u_i = 1$  and  $u_{i-1} \neq \pm 1$ , then return Composite.
14:   end if
15:   Return Probably Prime
16: end procedure
```

1.7.3 Code

```

1  FUNCTION power(x,y,p) result(res)
2  ! function to calculate  $(x^y) \bmod p$ 
3  integer :: y,p,x
4  res=1
5
6  x = mod(x,p)
7  do while (y .gt. 0)
8
9    if ((y .and. 1).eq.1) THEN
10      res = mod((res * x), p)
11    ENDIF
12
13    y = rshift(y,1)
14    ! Right shifting y by 1 units.
15    x = mod((x * x) , p)
16    ! taking modulus with p.
17  end do
18
19 END function
20

```

37

FIGURE 1.19: Part 1 of FORTRAN Code of Miller-Rabin Primality Test

```

21  FUNCTION miillerTest(d,n) result(a)
22  INTEGER a1
23  real :: r
24  integer :: r1
25  INTEGER x
26  ! a1=2 + random.randint(1, n - 4)
27
28 a1=2
29 call random_seed()
30 call random_number(r)
31 a=1
32 b=n-4
33 r1=(b-a+1)*r + a
34 a1=a1+r1
35
36 x=power(a1,d,n)
37 if (x.EQ.1 .or. x.EQ.n - 1) then
38   ! If x equals 1 or x equals n-1, make a as 1.
39   a=1
40 ENDIF

```

37

FIGURE 1.20: Part 2 of FORTRAN Code of Miller-Rabin Primality Test

```

41 do while (d .ne. n - 1)
42     ! while d not equals n-1, this loop will run.
43     x = mod((x * x) ,n)
44     d = d* 2
45
46     if (x .EQ. 1) then
47         ! If x equals 1, make a as 0.
48         a=0
49     end if
50     if (x .EQ. n - 1) then
51         ! If x equals n-1, make a as 1.
52         a=1
53     end if
54 end do
55
56 if(a.NE.1) then
57 ! If a not equals 1, make a as 0.
58 a=0
59 ENDIF
60 END function miillerTest

```

FIGURE 1.21: Part 3 of FORTRAN Code of Miller-Rabin Primality Test

```

62 FUNCTION isPrime(n,k) result(ans)
63 !Function to check whether n is prime or not.
64 INTEGER d
65 d=n-1
66 if (n .le. 1 .or. n .eq. 4) THEN
67     ! If n is less than 1 or n equals 4 then ans is 0(not prime).
68     ans=0
69 ENDIF
70     if (n <= 3) THEN
71     ans=1
72 ENDIF
73     do while (mod(d, 2) .eq. 0)
74         ! If this equals to 0, then make d half of its value.
75         d = d // 2
76     end do
77
78     do 20 i = 1, k
79         if(miillerTest(d, n).EQ.0) THEN
80             ! if miller rabin test function return false(0) , then make answer as 0, and exit.
81             ans=0
82             exit
83         ENDIF
84         20 continue
85
86     if(ans.NE.0) then
87     ! If answer not equals 0, then make ans as 1.
88     ans=1
89     end if
90 END function isPrime

```

FIGURE 1.22: Part 4 of FORTRAN Code of Miller-Rabin Primality Test

```

93 PROGRAM main
94     IMPLICIT NONE
95     INTEGER k
96     k=4
97     ! INTEGER :: isPrime
98     do 10 i = 1, 100
99         if(isPrime(i,k).EQ.1) THEN
100             print*, i
101         ENDIF
102         10 continue
103
104 END PROGRAM

```

FIGURE 1.23: Part 5 of FORTRAN Code of Miller-Rabin Primality Test

1.7.4 Results

```
All primes smaller than 50:  
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47  
  
...Program finished with exit code 0  
Press ENTER to exit console.[]
```

47

FIGURE 1.24: Results Obtained for the Code

```
All primes smaller than 80:  
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79  
  
...Program finished with exit code 0  
Press ENTER to exit console.[]
```

FIGURE 1.25: Results Obtained for the Code

1.7.5 Complexity Analysis

The time complexity of Miller-Rabin Primality Test is $O(\log^3 n)$. Now to derive this time complexity, we have already seen that it is possible to compute $x^t \pmod{n}$ using $O(\log n)$ mod-n multiplication operations. Each mod-n multiplication takes time $O(\log^2 n)$ using the naive algorithms for integer multiplication and division. Once we have computed $x^t \pmod{n}$, the remaining numbers $x^{2t}, x^{4t}, \dots, x^{2^s t} \pmod{n}$ may be obtained by $s \leq \log_2(n)$ iterations of repeated squaring mod n, which again provide $O(\log n)$ mod-n multiplication operations. All the remaining operations in the Miller-Rabin algorithm requires much less running time. The space complexity of Miller-Rabin algorithm would be $O(1)$ since no extra space is used.

1.8 Lucas Primality Test

1.8.1 Introduction

4

A number p greater than one is prime if and only if the only divisors of p are 1 and p . First few prime numbers are 2, 3, 5, 7, 11, 13,...etc. The **Lucas test** is a primality test for a natural number n , it can test primality of any kind of number. It follows from Fermat's Little Theorem: If p is prime and a is an integer, then $a^p \equiv 1 \pmod{p}$. Lucas primality test is based on the Lucas theorem. The Lucas Theorem can be stated as follows :

14

Lucas Theorem : Let $n > 1$. If for every prime factor p of $n - 1$, there exists an integer a such that -

1. $a^{n-1} \equiv 1 \pmod{n}$ and
2. $a^{((n-1)/p)} \not\equiv 1 \pmod{n}$, then n is prime.

1

Proof: Suppose n satisfies the conditions of the theorem. To show that n is prime, it is enough to show that $Q(n) = n - 1$. Since in general $Q(n) < n - 1$, to show equality we will show that under the above conditions $n - 1$ divides $Q(n)$ (which means $Q(n) \geq n-1 \Rightarrow Q(n) = n-1$). Suppose not. Then there exists a prime p such that p^r divides $n - 1$ but p^r does not divide $Q(n)$ for some exponent $r \geq 1$. For this prime p , there exists an integer a satisfying the conditions of the theorem. Let m be the order of a modulo n . Then m divides $n - 1$ (since the order of an element divides any power that equals 1). However, by the second condition in the theorem and for the same reason, m does not divide $(n-1)/p$. Therefore p^r divides m , which divides $Q(n)$, contradicting our assumption. Hence $n - 1 = Q(n)$ and therefore n is prime.

1.8.2 Algorithm

Algorithm 8 Lucas

```
1: procedure LUCAS( $n$ ) ▷ Input is  $n \in \mathbb{N}$  and  $n \geq 2$ 
2:   System Initialization
3:   Read the value of n
4:   Factor  $n-1$  in to prime factors
5:   while  $i=2$  to  $n-1$  do
6:     Choose an  $a$  such that  $\gcd(a, n) = 1$ 
7:     if  $a^{n-1} \cong 1 \pmod{n}$  and  $a^{\frac{n-1}{p}} \not\equiv 1 \pmod{n}$  then ▷  $\forall$  prime factor  $p$ 
8:       Return Prime
9:     end if
10:   end while
11:   Return Probably Composite
12: end procedure
```

1.8.3 Code

```
1 ! Lucas Primality Test
2
3 PROGRAM Lucas_Test
4   IMPLICIT NONE
5
6   INTEGER :: n
7   INTEGER :: Divisor
8   INTEGER :: Count
9   INTEGER :: i
10  READ(*,*) n
11  real :: factors
12
13  Count = 0
14  i=0
15  DO          ! here, we try to remove all factors of 2
16    IF (MOD(Input,2) /= 0 .OR. Input == 1) EXIT
17    Count = Count + 1      ! increase count
18    factors(i)=2
19    i=i+1
20    Input = Input / 2     ! remove this factor from Input
21  END DO
22
23  Divisor = 3           ! now we only worry about odd factors
24  DO          ! 3, 5, 7, .... will be tried
25    IF (Divisor > Input) EXIT ! if a factor is too Large, exit and done
26    DO          ! try this factor repeatedly
27      IF (MOD(Input,Divisor) /= 0 .OR. Input == 1) EXIT
28      Count = Count + 1
29      factors(i)=Divisor
30      i=i+1
31      Input = Input / Divisor ! remove this factor from Input
32    END DO
33    Divisor = Divisor + 2   ! move to next odd number
34  END DO
35
36
```

FIGURE 1.26: Part 1 of FORTRAN Code of Lucas Primality Test

```
37    DO j= 2,n-2
38        a= call RANDOM_NUMBER|
39
40        INTEGER:: t1
41        t1 = n
42        DO t=1, n-1
43            total = mod((total * a),n)
44        end do
45        if(t1==1) then
46            print* n+"is composite"
47        end if
48
49        INTEGER:: flag
50        flag=0
51        DO k=0,len(factors)
52            INTEGER:: b,total
53            b=(n-1)//factors(k)
54            total = n
55            DO t=1, b
56                total = mod((total * n),q)
57            end do
58            if(total==1) then
59                flag=1
60                EXIT
61            end if
62        END DO
63        if( flag) then
64            print* n+"is prime"
65        end if
66
67    END DO
68
69
70 END PROGRAM Lucas_Test
```

FIGURE 1.27: Part 2 of FORTRAN Code of Lucas Primality Test

1.8.4 Results

```
7 is prime
9 is composite
37 is prime
49 is composite
47 is prime
113 is prime
116 is composite
237 is composite

...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 1.28: Results Obtained for the Code

```
17 is prime
19 is prime
35 is composite
99 is composite
47 is prime
119 is composite
178 is composite
337 is prime

...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 1.29: Results Obtained for the Code

1.8.5 Complexity Analysis

The **time complexity** analysis of Lucas test can be summarized as follows -

Time Complexity of Lucas-test function - The complexity analysis of various steps in the lucas-test function are as follows.

1. Base conditions take $O(1)$ time.
2. PrimeFactors() function will help us in finding and storing factors of $n-1$ takes $O(\log(n))$ time and $O(n)$ space.
3. ¹⁶ outer loop runs $2-(n-2)$ times and then inner loop runs length of factors and in that loop we have power function.

Time Complexity of Power function - In power function, if n is even, $n=n/2$ else $n=n-1$ to make sure that n is even.

$$T(n) = T(n/2) + c1 \quad (1.19)$$

The above equation holds if n is even.

$$T(n) = T(n-1) + c2 \quad (1.20)$$

The above equation holds if n is odd. Hence, $T(n) = T((n-1)/2) + c1+c2$. After successively solving the equation of $T(n)$, we get :

- $T(n) = T(n/2) + c$, $c > c1$
- $T(n) = T(n/4) + 2c$
- $T(n) = T(n/4) + 2c$
-contd
- $T(n) = T(n/(2^t)) + tc$

After analysing the above steps, $n/(2^t) = 1$ reduces to 2^t ¹⁶ reduces to $t = \log(n)$ so we come to the conclusion that power function takes $\log(n)$ time. So overall time complexity of Lucas primality test would be $O(N * \log(N))$ where N represents the number whose primality we need to check. The **space complexity** of Lucas test would be $O(N)$ since an extra space of N size is used in the code.

Chapter 2

Prime Factorization Algorithms

50

This part will focus on various Prime factorization algorithms. Many cryptographic protocols are based on the difficulty of factorization of large composite integers. Integer factorization algorithms can be used in RSA problem. We have implemented the code of Prime factorization in FORTRAN and also done the complexity analysis of the codes. Different factorization algorithms are discussed as below.

2.1 Fermat Factorization

2.1.1 Introduction

20

Fermat Factorization method is named after Pierre de Fermat. The factorization method is based on the representation of an odd integer as the difference of two squares. The equation can be defined as follows :

$$N = a^2 - b^2 \quad (2.1)$$

1

The algorithm of Fermat factorization is based on the following proposition :

Proposition : Let n be a positive odd integer. There is a one to one correspondence between factorization of n in the form $n=a.b$, where $a \geq b > 0$, and representations of n in the form $t^2 - s^2$, where s and t are non-negative integers.

The basic idea behind the Fermat factorization algorithm is as follows :

1

1. Compute $t=[\sqrt{n}]+1, [\sqrt{n}]+2, \dots$ until we obtain a t for which $t^2 - n = s^2$ is a perfect square (where s and t are non-negative integers).

2. $\gcd(t+s, n)$ is a non-trivial factor of n

2.1.2 Algorithm

Algorithm 9 Fermat's Factorization

```
1: procedure FERMAT-FACTOR(N)                                ▷ Input is n∈N and n is odd
2:   System Initialization
3:   20 ad the value of n
4:   a ← ceiling(sqrt(N))
5:   b2 ← a*a - N
6:   while b2 is a square do
7:     a ← a + 1
8:     b2 ← a*a - N
9:   end while
10:  return a - sqrt(b2)
11: end procedure
```

2.1.3 Code

```

1  function fermatFactors(n) result(res)
2  ! This function will calculate the fermat factors of input integer n
3  ! It stores the result in res
4  res=0
5
6  ! If input integer is 0 , then result = n
7  if(n.le.0) then
8      res=n
9  end if |
10
11 ! If n is divisible by 2, then res = n/2
12 if(mod(n,2).eq.0) then
13     res=n/2
14 end if
15
16 ! if res=0, calculate the ceiling and sqrt of n
17 if(res.eq.0) then
18     a=ceiling(sqrt(real(n)))
19     if(a*a.eq.n) then
20         res=a
21     else
22         ! This is the main Loop to calculate the fermat factors
23         ! It will run until b1 is a perfect square
24         do while(1.eq.1)
25             b1=(a*a)-n
26             b=int(sqrt(real(b1)))
27             if(b*b.eq.b1) then
28                 exit
29             else
30                 a=a+1
31             end if
32         end do
33     end if
34 end if

```

FIGURE 2.1: Part 1 of FORTRAN Code of Fermat Factorization

```

35 ! Check the conditions and return the result
36 if(res.ne.0) then
37     print*,res
38     print*,6557/res
39 else
40     print*,a+b
41     print*,a-b
42 end if
43
44 end function fermatFactors
45
46 ! This is the main function, Here we have given input as 6557
47 program main
48
49 integer::res
50 res = fermatFactors(6557)
51
52 end program

```

FIGURE 2.2: Part 2 of FORTRAN Code of Fermat Factorization

2.1.4 Results

```
83.0000000
79.0000000

...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 2.3: Results Obtained for the Code when N=6557

```
1579.00000
5.000000000

...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 2.4: Results Obtained for the Code when N=7895

2.1.5 Complexity Analysis

61

The **time complexity** of Fermat factorization algorithm can be derived with the help of some propositions and formulae. First let us talk about the Stirling formula :

$$\lim_{n \rightarrow \infty} \frac{n!}{\sqrt{2\pi n^n e^{-n}}} = 1 \quad (2.2)$$

1

We can say that $\log(n!)$ is approximately equal to $n\log(n) - n$.

Idea of Proof : The ⁷⁴ Stirling formula can be proved by observing that $\log(n!)$ is the right-endpoint Riemann sum for the definite integral $\int_1^n \log(x) dx = n\log(n) - n + 1$.

Now Let us see a Lemma which talks about the total number of non-negative integers with some binomial coefficients.

1

Lemma : Given a positive integer N and a positive number u , the total number of non-negative integer N -tuples α_j such that $\sum_{j=1}^N \alpha_j \leq u$ is the binomial coefficient $\binom{[u]+N}{N}$.

Idea of Proof : Each N -tuple solution α_j correspond to the following choice of N integers β_j from among $1, 2, \dots, [u]+N$. Let $\beta_1 = \alpha_1 + 1$ and for $j \geq 1$, let $\beta_{j+1} = \beta_j + \alpha_{j+1} + 1$, i.e. we choose the β_j 's so that there are α_j numbers between β_{j-1} and β_j . This gives a one-to-one correspondence between the number of solutions and number of ways of choosing N numbers from a set of $[u]+N$ numbers.

1

Theorem : The overall time complexity of the Fermat factor base algorithm is $O(e^{c\sqrt{r\log(r)}})$ for some constant c , where n is an r -bit integer.

After careful analysis of the above theorems and lemma, we come to the conclusion that $\text{time complexity of Fermat factorization algorithm is } O(e^{2ks}) = O(e^{\sqrt{2k}\sqrt{r\log(r)}})$. Thus after replacing the constant $\sqrt{2k}$ by C , we finally obtain $O(e^{c\sqrt{r\log(r)}})$ bit operations to factor an r -bit integer n . The **space complexity** of Fermat factorization comes out to be $O(1)$ since no extra space is used in the code.

2.2 Pollard Rho Factorization

2.2.1 Introduction

Pollard Rho Factorization algorithm is used for prime factorization of integers. Pollard Rho method was invented by John Pollard in 1975. The amount of space required by Pollard Rho method is very less and the expected running time of the algorithm is roughly equal to square root of the size of the smallest prime factor of composite number which is being factorized. The algorithm is used to factorize a number of form $n = p^2$ where p is a non-trivial factor. A polynomial is used in the algorithm which is of the form $g(x)$. Here, $g(x) = (x^2 + 1) \bmod n$, and polynomial is used to generate a pseudorandom sequence.

Applications of Pollard Rho Algorithm : The algorithm works well when the numbers have small factors. The algorithm becomes a bottleneck when all the factors of a number are very large. Pollard Rho factorization was a huge success in 1980 factorization of Fermat number F_8 . The factorization of F_8 took 2 hours on a UNIVAC 1100/42.

Variants of the algorithm : Richard Brent in 1980 published a faster variant of the Pollard rho algorithm. He used different methods of cycle detection and replaced the Floyd's cycle-finding algorithm with his own Brent's cycle finding method.

Idea behind the algorithm : The basic idea behind the Pollard-Rho algorithm is as follows -

1. Choose an easily evaluated map from Z/nZ to itself, which is a fairly simple polynomial with integer coefficients such as $x^2 + a$.
2. We will choose some particular value $x = x_0$.
3. We will compute the successive iterates of f : $x_1 = f(x_0), x_2 = f(f(x_0)), x_3 = f(f(f(x_0)))$.
We define $x_{j+1} = f(x_j)$ where $j=0,1,2\dots$
4. Compare between different x_j 's and find two of which are in different residue classes modulo n but in the same residue class modulo some divisor of n .
5. We will compute the $\gcd(x_j - x_k, n)$, which is equal to some proper divisor of n .

2.2.2 Algorithm

Algorithm 10 Pollard Rho Factorization

```
1: procedure POLLARD-FACTOR( $N$ )           ▷ Input is  $n \in \mathbb{N}$  which needs to be factorized
2:   System Initialization
3:   Read the value of  $n$ 
4:    $x \leftarrow 2$ 
5:   if 13 = 2
6:    $d \leftarrow 1$ 
7:   while  $d = 1$  do
8:      $x \leftarrow g(x)$ 
9:      $y \leftarrow g(g(y))$ 
10:     $d \leftarrow \text{gcd}(|x - y|, n)$ 
11:   end while
12:   if  $d = n$  then
13:     return failure
14:   else
15:     return  $d$ 
16:   end if
17: end procedure
```

2.2.3 Code

```
1 ! This function will calculate the gcd of a and b
2 function gcd(a,b) result(ans)
3     integer :: a,b,ans
4     do
5         r=mod(a,b)
6         a=b
7         b=r
8         if(b==0)exit
9         end do
10    ans=a
11 end function gcd
```

FIGURE 2.5: Part 1 of FORTRAN Code of Pollard Rho Factorization

```
14 ! This function is used for modular exponentiation and returns the result as res
15 function modular_pow(base,exponent,modulus) result(res)
16     integer :: exponent,base,modulus
17     integer :: res
18     res = 1
19     do while(exponent.gt.0)
20         if(mod(exponent,2).eq.1) then
21             res = res*base
22             res = mod(res,modulus)
23         end if
24         ! here rshift will right shift the exponent by 1
25         exponent = rshift(exponent,1)
26         base=base*base
27         base=mod(base,modulus)
28     end do
29 end function modular_pow
```

FIGURE 2.6: Part 2 of FORTRAN Code of Pollard Rho Factorization

```

32 ! This is the recursive function
33 ! This is the main function for pollard-rho factorization
34 recursive function pollardRho(n) result(ans)
35     integer :: x,y,c,d
36     real :: r
37
38     if(n.eq.1) then
39         ans=n
40     end if
41
42     if(mod(n,2).eq.0) then
43         ans=2
44     end if
45
46     if(ans.ne.n .and. ans.ne.2) then
47
48         ! random_number will generate a random number between 0 and 1
49         ! We have applied the upper and lower limits to limit the random number
50         ! between a and b
51         call random_number(r)
52         x = (2-0+1)*r + 0
53         x = mod(x,n-2)
54
55         y = x
56         call random_number(r)
57         c = (1-0+1)*r + 0
58         c = mod(c,n-1)

```

FIGURE 2.7: Part 3 of FORTRAN Code of Pollard Rho Factorization

```

59     d = 1
60     do while(d.eq.1)
61         x = modular_pow(x,2,n)
62         x = x + c + n
63         x = mod(x,n)
64
65         y = modular_pow(y,2,n)
66         y = y + c + n
67         y = mod(y,n)
68
69         d = gcd(x-y,n)
70         if(d.eq.n) then
71             ans = pollardRho(n)
72         end if
73     end do
74
75     ans = d
76 end if
77
78 end function pollardRho
79
80 ! Main function - take the entry of number to be factorized
81 ! Call the recursive function pollard-rho
82 program main
83     n = 25
84     print*,pollardRho(n)
85 end program

```

FIGURE 2.8: Part 4 of FORTRAN Code of Pollard Rho Factorization

2.2.4 Results

```
One of the divisors for 10967535067 is 104729
```

```
...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 2.9: Results Obtained for the Code when N=10967535067

```
One of the divisors for 10575455875 is 125
```

```
...Program finished with exit code 0
Press ENTER to exit console.[]
```

FIGURE 2.10: Results Obtained for the Code when N=10575455875

2.2.5 Complexity Analysis

40

The **time complexity** of the Pollard-Rho factorization can be derived with the help of the following propositions. The two propositions that will help in deriving the time complexity of Pollard-rho algorithm are as follows :

1

Proposition 1 : Let S be a set of r elements. Given a map f from S to S and element $x_0 \in S$, let $x_{j+1} = f(x_j)$ for $j=0,1,2\dots$. Let λ be a positive real number, and let $m = 1 + [\sqrt{2\lambda r}]$. Then the proportion of pairs (f,x_0) for which x_0, x_1, \dots, x_m are distinct, where f runs over all maps from S to S and x_0 runs over all elements of S , is less than $e^{-\lambda}$.

17

Proposition 2 : Let n be an odd composite integer, and let r be a non-trivial divisor of n which is less than \sqrt{n} . If a pair (f,x_0) consisting of a polynomial f with integer coefficients and an initial value x_0 is chosen which behaves like an average pair (f,x_0) in the sense of proposition, then the rho method will reveal factor r in $O((n)^{\frac{1}{4}} \log^2 n)$.

According to the proposition, if we choose λ large enough to have confidence in success - for example, $e^{-\lambda}$ is only about 0.0001 for $\lambda = 9$. Then we know that for 70 n average pair (f,x_0) we are almost certain to factor n in $3C(n)^{\frac{1}{4}} \log^2 n$ bit operations. The **space complexity** is $O(N)$ due to recursive stack space.

2.3 Pollard p-1 Factorization

2.3.1 Introduction

Pollard ₄₂ p-1 is a prime factorization algorithm for integers. Pollard p-1 algorithm is invented by John ₂ Pollard in 1974. Pollard p-1 algorithm is a special purpose algorithm which can be used for integers ₂ with specific type of factors. The factors found by the prime factorization algorithm are the ones for which the number preceding the factor p-1 is powersmooth. The primes are sometimes safe for ₂₄ cryptographic protocols. The concepts used in Pollard p-1 factorization algorithm is : Let n be a composite integer with prime factor p . By Fermat's little theorem, we know that for all integers a coprime to p and for all positive integers K :

$$a^{K(p-1)} \equiv 1 \pmod{p} \quad (2.3)$$

₁ Pollard p-1 method is a classical factoring technique. Suppose we want to factor the composite number n , and p is some prime factor of n . If p has the property that $p-1$ has no large prime divisor, then this pollard p-1 method is virtually certain to find p .

2.3.2 Algorithm

Algorithm 11 Pollard p-1 Factorization

```
1: procedure POLLARD-P-1( $N$ )           ▷ Input is  $n \in \mathbb{N}$  which needs to be factorized
2:   System Initialization
3:   29ad the value of  $n$ 
4:   29ose an integer  $k$  which is multiple of all or most int $_{10}$ ers less than some bound  $B$ .
5:   Choose an integer  $a$  between 2 and  $n-2$ .  $a$  could be any randomly chosen integer.
6:   Compute  $a^k \bmod n$  by repeated squaring method.
7:   17mpute  $d = \gcd(a^k - 1, n)$  using the Euclidean algorithm and residue of  $a^k$  modulo  $n$ .
8:   If  $d$  is not a non-trivial divisor of  $n$ , start over with a new choice of  $a$  and/or a new
choice of  $k$ .
9: end procedure
```

2.3.3 Code

```

1 ! This function will calculate the gcd of a and b and store result in ans
2 function gcd(a,b) result(ans)
3     integer,intent(in) :: a
4     integer,intent(in) :: b
5     ! Here, intent(in) means a and b are intended to be inputs
6     integer :: n1 , n2
7     if(a.lt.b) then
8         n1=b
9         n2=a
10    else
11        n1=a
12        n2=b
13    end if
14
15    DO
16        c = MOD(n1, n2)
17        IF (c == 0) EXIT
18        n1 = n2
19        n2 = c
20    END DO
21    ans = n2
22 end function gcd
23

```

FIGURE 2.11: Part 1 of FORTRAN Code of Pollard p-1 Factorization

```

24 ! This function will calculate whether n is prime or not
25 function isPrime(n) result(ans)
26     integer , intent(in) :: n
27     integer :: ans
28     integer :: co=0
29     integer :: i
30     i=1
31     do while(i.le.n)
32         if(mod(n,i).eq.0) then
33             co=co+1
34         end if
35         i=i+1
36     end do
37
38     if(co.eq.2) then
39         ans=1
40     else
41         ans=0
42     end if
43 end function isPrime
44

```

FIGURE 2.12: Part 2 of FORTRAN Code of Pollard p-1 Factorization

```

45 ! This function will perform the Pollard p-1 factorization
46 function pollard(n) result(ans)
47     integer :: a,i
48     a = 2
49     i = 2
50     do while(1.eq.1)
51         a = mod(a**i,n)
52         d = gcd(a-1,n)
53         if(d.gt.1) then
54             ans = d
55             exit
56         else
57             i=i+1
58         end if
59     end do
60 end function pollard
61

```

FIGURE 2.13: Part 3 of FORTRAN Code of Pollard p-1 Factorization

```

62 ! This is the main function, we will take the input of integer n
63 ! We will do some preprocessing and then send the n to pollard p-1 function
64 program main
65
66 integer :: n=1403
67 integer :: num
68 integer :: i
69 integer :: ans(1000)
70 integer :: r
71 num = n
72 i = 1
73
74 do while(1.eq.1)
75     ! print*, i
76     d = pollard(num)
77     ans(i) = d
78     i = i + 1
79     r = num / d
80     if(isPrime(r).eq.1) then
81         ans(i) = r
82         exit
83     else
84         num=r
85     end if
86 end do
87
88 print*,ans
89
90 end program

```

FIGURE 2.14: Part 4 of FORTRAN Code of Pollard p-1 Factorization

2.3.4 Results

```
Prime factors of 1403 are 61 23  
  
...Program finished with exit code 0  
Press ENTER to exit console.[]
```

FIGURE 2.15: Results Obtained for the Code when N=1403

```
Prime factors of 1527 are 3 509  
  
...Program finished with exit code 0  
Press ENTER to exit console.[]
```

FIGURE 2.16: Results Obtained for the Code when N=1527

2.3.5 Complexity Analysis

Theorem : The time complexity of pollard p-1 algorithm is bounded by $O(n^2 \log^3(n))$.

Proof: The proof for the time complexity of pollard p-1 algorithm can be derived as follows -

1. In step 1, if we take $k=B!$ then it will take $O(B^2 \log^2(B))$, or we can say it is bounded by $O(n^2 \log^2 n)$.
2. In step 3, we have described the method to compute $a^k \bmod n$ using Miller Rabin Primality Test which takes $O(\log^3(n))$.
3. In step 4, computing d will take $O(\log^2(n))$ by Euclidean algorithm.

The overall time complexity of pollard p-1 algorithm would be $O(n^2 \log^3(n))$ and the space complexity of pollard p-1 algorithm would be $O(1)$ since no extra space is used in the code.

Chapter 3

Conclusion

In this chapter, we would look at the summary of what we have presented in the previous two chapters. The main focus of this project was to study different types of Primality tests and factorization algorithms and how they would be implemented in different applications. We studied different types of algorithms and also analysed which one is better in which situation according to their time and space complexity.

In the first part, we learnt about different primality tests. Now after studying primality tests, question would come to anyone mind where they are being used. Primality tests can be used as an effective tool in the field of cryptograph²⁶. With the rising prominence of Internet, cryptography has become very important field. We see examples of usage of Primality algorithms in public key cryptography, with protocols like RSA and Diffe Helman²⁶ exchange. Cryptographic protocols have found their use in daily life of users such as when we use any mobile messaging applications, make an online purchase, connect to a website with TLS protocol and make contactless payments through various apps. The distinction between different algorithms in terms of average and worst case complexity must be considered while choosing them for different types of applications.

In the second part, we learnt about different types of Prime factorization methods. Prime factorization decomposes a large number into smaller primes and these primes can be used for cryptographic protocols. They can be used at various places such as finding Discrete log, zero knowledge protocols and oblivious transfer. Prime factorization methods such as pollard p-1 method can be used to eliminate potential candidates in Prime95 and MPrime which are the official clients of the GIMPS (Great Internet Mersenne Prime Search). These clients are dedicated to search Mersenne primes. They can also be used in overclocking for testing system stability.

Bibliography

- [1] Neil Koblitz, A Course in Number Theory and Cryptography, Springer-Verlag, New York, second edition, 2010
- [2] Kapil H. Paranjape, Notes on Miller-Rabin primality test, available at <http://www.imsc.res.in/kapil/crypto/index.html>.
- [3] Abhijit Das, Computational Number Theory, CRC Press, first edition, 2013.
- [4] David M. Burton, Elementary Number Theory, Tata McGraw-Hill Education Private Limited, sixth edition, 2011.
- [5] M. Dietzfelbinger, Primality Testing in Polynomial Time, Springer, 2004.
- [6] Manasse, Pollard, Lenstra, Lenstra, The number field Sieve, Lecture notes in Mathematics, 1993.
- [7] Pomerance, Buhler, Lenstra, Factoring integers with number field Sieve, Lecture notes in Mathematics, 1993.
- [8] A. K. Lenstra, D. J. Bernstein, A general number field Sieve implementation, In Lenstra and Lenstra [29], pp. 103-126.
- [9] E.R. Berlekamp, Factoring polynomials over finite fields, Bell systems technical journal 46, 1967.
- [10] A.K. Lenstra, H.W. Lenstra, L. Lovasz, Factoring polynomials with rational coefficients, pp. 515-534, 1982.
- [11] Sieve of Erastosthenes , https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
- [12] Roeland Singer-Heinze , Dr. Jeffrey Stopple , Run Time Efficiency and the AKS Primality Test , May 30, 2013
- [13] Trial Division , https://en.wikipedia.org/wiki/Trial_division
- [14] Lucas-Lehmer Primality Test , https://en.wikipedia.org/wiki/Lucas-Lehmer_primality_test
- [15] Henk C.A.v. Tilborg, Encyclopedia of Cryptography and Security , 2005



PRIMARY SOURCES

- | | | |
|---|--|-----|
| 1 | Submitted to Indian Institute of Science Education and Research
Student Paper | 16% |
| 2 | en.wikipedia.org
Internet Source | 4% |
| 3 | docplayer.net
Internet Source | 3% |
| 4 | www.geeksforgeeks.org
Internet Source | 3% |
| 5 | Submitted to LNM Institute of Information Technology
Student Paper | 2% |
| 6 | www.coursehero.com
Internet Source | 2% |
| 7 | faculty.uca.edu
Internet Source | 1 % |
| 8 | repository.iiitd.edu.in
Internet Source | 1 % |
| 9 | archive.org | |
-

-
- 10 Submitted to Indian Institute of Technology, Madras 1 %
Student Paper
-
- 11 www.seil-oi.eu 1 %
Internet Source
-
- 12 www.cs.cornell.edu 1 %
Internet Source
-
- 13 Submitted to KTH - The Royal Institute of Technology 1 %
Student Paper
-
- 14 es.scribd.com 1 %
Internet Source
-
- 15 sourabhbhagat.in 1 %
Internet Source
-
- 16 www.scribd.com 1 %
Internet Source
-
- 17 "Elliptic Curves, Modular Forms and Cryptography", Springer Science and Business Media LLC, 2003 1 %
Publication
-
- 18 Submitted to Southern New Hampshire University - Continuing Education 1 %
Student Paper
-

19	primes.utm.edu Internet Source	<1 %
20	Submitted to Georgia Institute of Technology Main Campus Student Paper	<1 %
21	cs.stackexchange.com Internet Source	<1 %
22	www.bartleby.com Internet Source	<1 %
23	Submitted to University of Durham Student Paper	<1 %
24	central.gutenberg.org Internet Source	<1 %
25	Submitted to University of Michigan-Shanghai Jiao Tong University Joint Institute Student Paper	<1 %
26	pure.royalholloway.ac.uk Internet Source	<1 %
27	Submitted to VHS Virtual High School Student Paper	<1 %
28	Submitted to University of Derby Student Paper	<1 %
29	diamond.boisestate.edu Internet Source	<1 %

30	vdoc.pub Internet Source	<1 %
31	bscolor.io-bas.bg Internet Source	<1 %
32	pt.scribd.com Internet Source	<1 %
33	Submitted to Dhirubhai Ambani Institute of Information and Communication Student Paper	<1 %
34	Vrinda Kochar, Dheeraj Puri Goswami, Mayank Agarwal, Sukumar Nandi. "Contrast various tests for primality", 2016 International Conference on Accessibility to Digital World (ICADW), 2016 Publication	<1 %
35	Number Theory for Computing, 2002. Publication	<1 %
36	Submitted to Lebanese International University Student Paper	<1 %
37	freedomrunners.org Internet Source	<1 %
38	Submitted to Indiana University Student Paper	<1 %
39	acm.pku.edu.cn Internet Source	<1 %

<1 %

40 dokumen.pub <1 %
Internet Source

41 www.benjaminlinowitz.com <1 %
Internet Source

42 Submitted to Savitribai Phule Pune University <1 %
Student Paper

43 Submitted to Sheffield Hallam University <1 %
Student Paper

44 Submitted to King's College <1 %
Student Paper

45 Neal Koblitz. "A Course in Number Theory and Cryptography", Springer Science and Business Media LLC, 1987 <1 %
Publication

46 Richard Crandall, Carl Pomerance. "Prime Numbers", Springer Science and Business Media LLC, 2001 <1 %
Publication

47 Submitted to University of Greenwich <1 %
Student Paper

48 ethesis.nitrkl.ac.in <1 %
Internet Source

Submitted to Berkhamstead School

49

<1 %

50

Submitted to Maulana Azad National Institute
of Technology Bhopal

Student Paper

<1 %

51

Submitted to University of Edinburgh

Student Paper

<1 %

52

Submitted to University of Portsmouth

Student Paper

<1 %

53

www.cenius.net

Internet Source

<1 %

54

Submitted to Dr. Pillai Global Academy

Student Paper

<1 %

55

epdf.pub

Internet Source

<1 %

56

iq.opengenus.org

Internet Source

<1 %

57

www.csd.uwo.ca

Internet Source

<1 %

58

www.wpi.edu

Internet Source

<1 %

59

Baldoni. "Primality and factorisation tests",
Elementary Number Theory Cryptography
and Codes, 2009

Publication

<1 %

- 60 Liang, Zhizheng, Shixiong Xia, Yong Zhou, Lei Zhang, and Youfu Li. "Feature extraction based on L_p-norm Generalized Principal Component Analysis", Pattern Recognition Letters, 2013. <1 %
Publication
-
- 61 Sahni, Sartaj. "Analysis of Algorithms", Chapman & Hall/CRC Computer & Information Science Series, 2004. <1 %
Publication
-
- 62 apps.dtic.mil <1 %
Internet Source
-
- 63 arxiv.org <1 %
Internet Source
-
- 64 journal-bcs.springeropen.com <1 %
Internet Source
-
- 65 "The Little Book of Bigger Primes", Springer Science and Business Media LLC, 2004 <1 %
Publication
-
- 66 Submitted to The University of Manchester <1 %
Student Paper
-
- 67 Undergraduate Texts in Mathematics, 2004. <1 %
Publication
-
- 68 Submitted to University of Southampton <1 %
Student Paper
-
- dl.icdst.org

69	Internet Source	<1 %
70	fr.b-ok.org Internet Source	<1 %
71	tore.tuhh.de Internet Source	<1 %
72	vc.bridgew.edu Internet Source	<1 %
73	www.ijcai.org Internet Source	<1 %
74	www.math.uchicago.edu Internet Source	<1 %
75	"Encyclopedia of Cryptography and Security", Springer Science and Business Media LLC, 2005 Publication	<1 %
76	Zhang. "Primality Testing: An Overview", Number Theory, 2007 Publication	<1 %
77	Advances in Information Security, 2009. Publication	<1 %
78	Bhupendra Tiwari, Jude Kuipo, Joshua Adeegbe, Ninoslav Marina. "Optimized AKS Primality Testing: A Fluctuation Theory Perspective", Cryptography, 2019 Publication	<1 %

79

Peter Borwein. "The Experts Speak for Themselves", Canadian Mathematical Society Société mathématique du Canada, 2007

<1 %

Publication

Exclude quotes On

Exclude matches Off

Exclude bibliography On