Factorization into primes and their applications

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

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

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.[1]

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 probablistic primality test is a nondeterministic primality test which returns whether the input integer n is not a prime or it 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.[1][2]

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.[1]

Contents

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

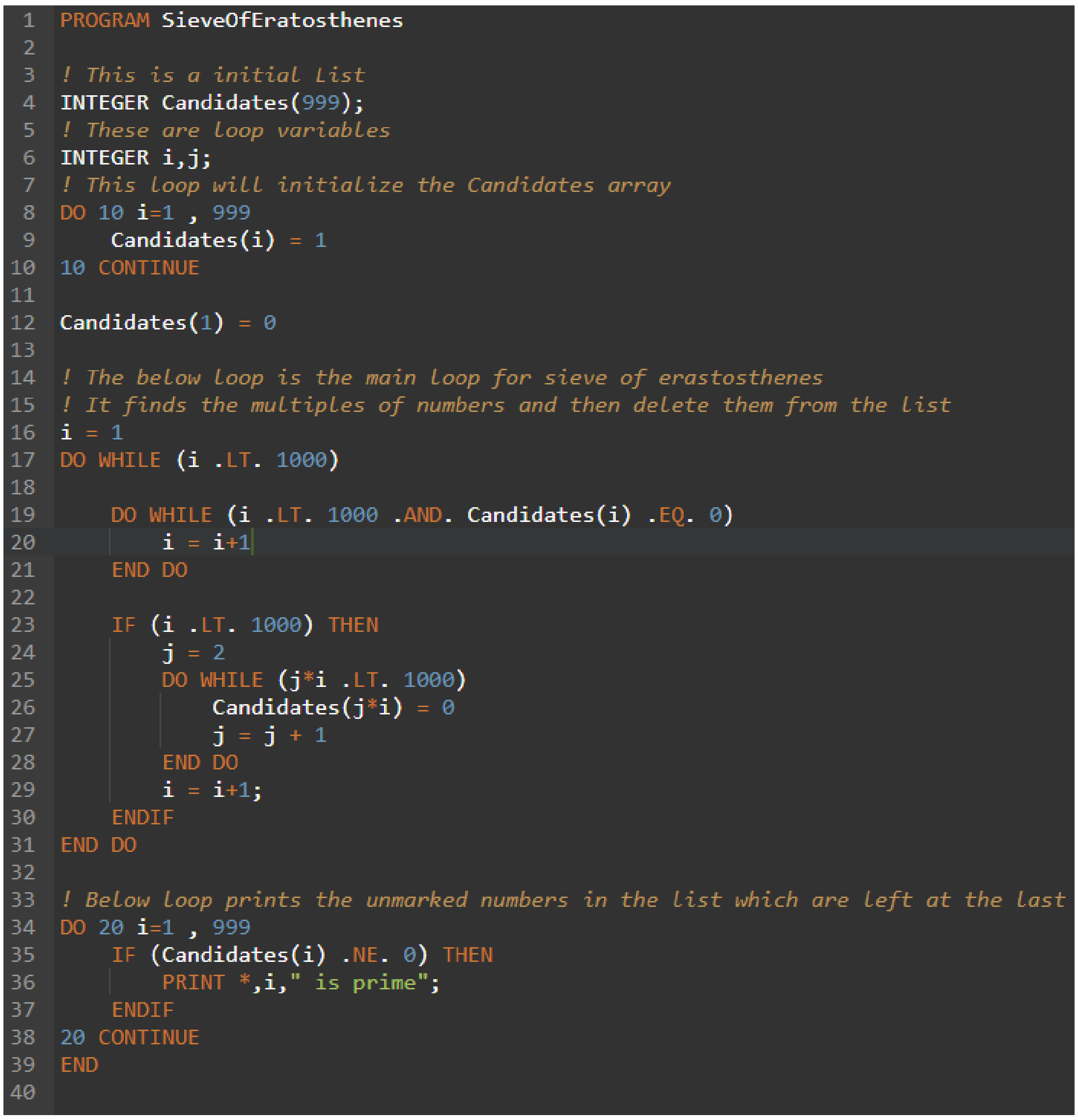
The **Sieve of Erastosthenes** is an ancient algorithm for finding the prime numbers upto any given limit. The sieve of erastosthenes is one of the effiecient ways to find all primes smaller than a given input integer n. The idea behind the algorithm is simple, first it creates a table of numbers from 2 to given integer n. Then it takes the prime numbers one at a time and remove their multiples from the table. It continues the process until p $<=$ $\sqrt{n}$ where p is a prime number.
The earliest known reference 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 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
    System Initialization
    Read the value of n
    a[1....n] integer array
    while j=1 to n do
        a[j] $\leftarrow$ j
    end while
    i $\leftarrow$ 2
    while $i^2$ < n do
        if a[i] $\neq$ 0 then
            t $\leftarrow$ 2.i
            while t $\leq$ n do
                a[i] $\leftarrow$ 0
                t $\leftarrow$ t + i
            end while
        end if
    end while
    i $\leftarrow$ i + 1
    while j = 2 to n do
        if a[j] $\neq$ 0 then
            return a[j] is prime
        end if
    end while
end procedure
```

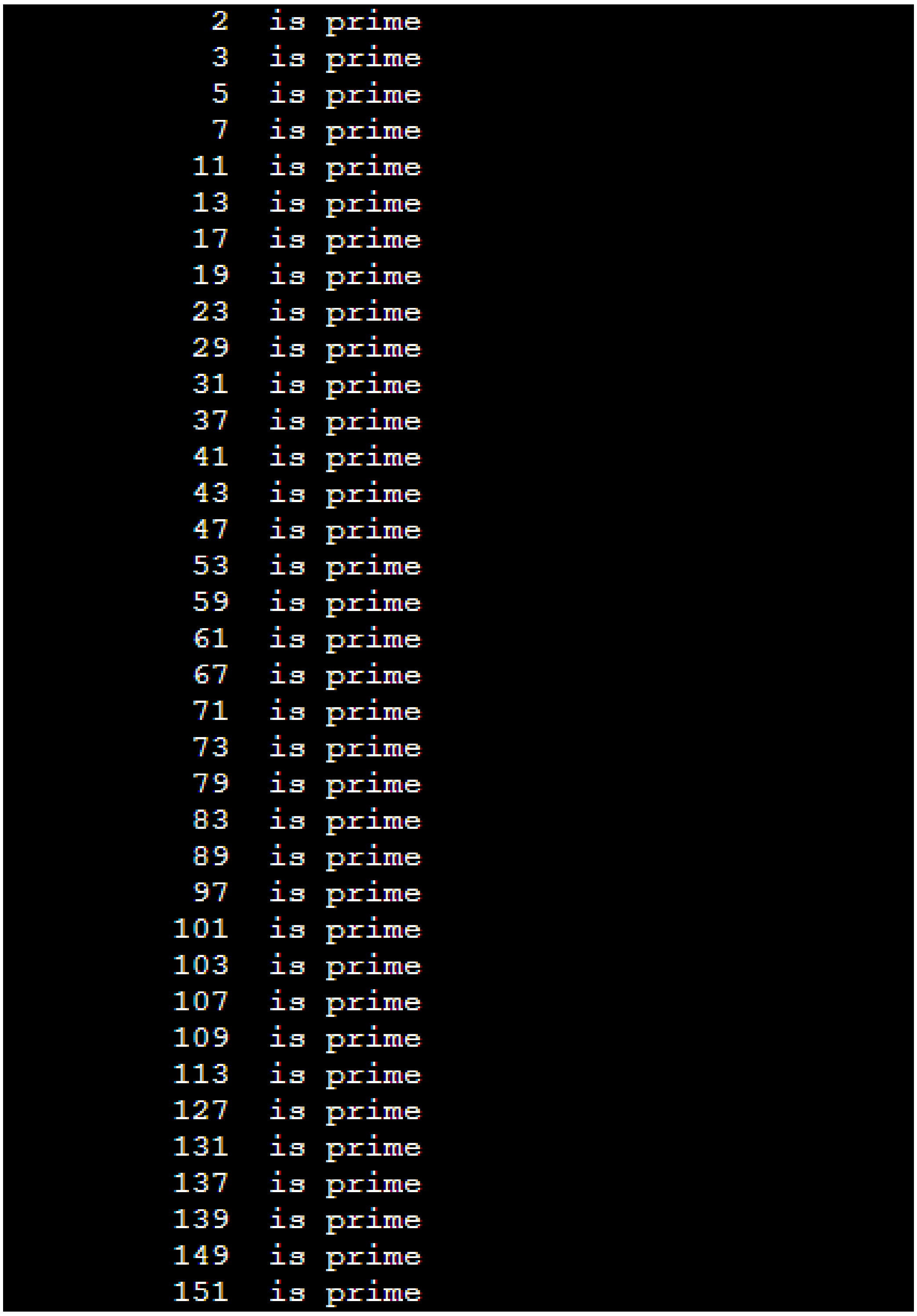
1.1.3 Code



```fortran
PROGRAM SieveOfEratosthenes

! This is a initial List
INTEGER Candidates(999);
! These are loop variables
INTEGER i,j;
! This loop will initialize the Candidates array
DO 10 i=1 , 999
    Candidates(i) = 1
10 CONTINUE

Candidates(1) = 0

! The below loop is the main loop for sieve of erastosthenes
! It finds the multiples of numbers and then delete them from the list
i = 1
DO WHILE (i .LT. 1000)

    DO WHILE (i .LT. 1000 .AND. Candidates(i) .EQ. 0)
        i = i+1
    END DO

    IF (i .LT. 1000) THEN
        j = 2
        DO WHILE (j*i .LT. 1000)
            Candidates(j*i) = 0
            j = j + 1
        END DO
        i = i+1;
    ENDIF
END DO

! Below loop prints the unmarked numbers in the list which are left at the last
DO 20 i=1 , 999
    IF (Candidates(i) .NE. 0) THEN
        PRINT *,i," is prime";
    ENDIF
20 CONTINUE
END
```

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 [3] :

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 -

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + ...p \tag{1.1}$$

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

$$n * (\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7}....\infty) \tag{1.2}$$

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

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

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

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

5. The **space complexity** of Sieve of Erastosthenes would be **O(N)** where N would be the size of the candidates array in the code.

1.2 AKS Primality Test

1.2.1 Introduction

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 [4] [5].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 [6] :

$$x + a^n \cong (x^n + a)(modn) \tag{1.5}$$

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:

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

which can be expressed in simpler term as :

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

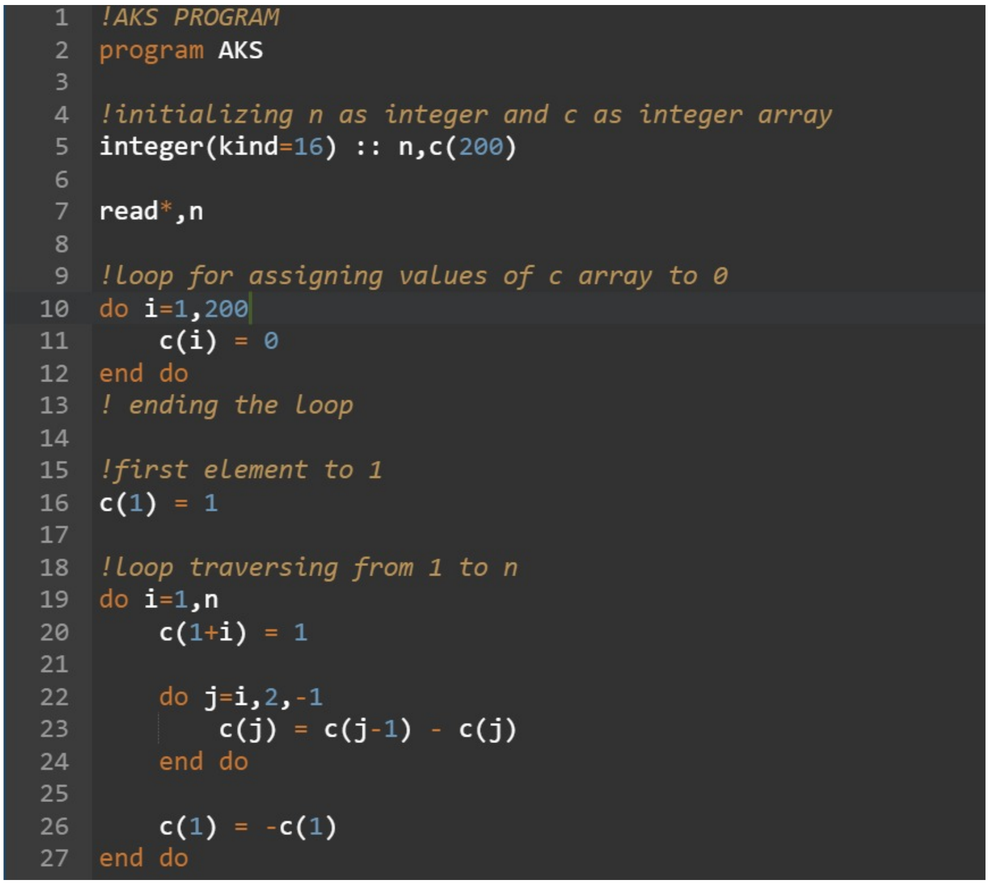
for some polynomials f and g .
This congruence can be checked in polynomial time when r is polynomial to the digits of n. The AKS algorithm evaluates this congruence for a large set of a values, whose size is polynomial to the digits of n. The proof of validity of the AKS algorithm shows that one can 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. [6]

1.2.2 Algorithm

Algorithm 2 AKS Primality Test [1]

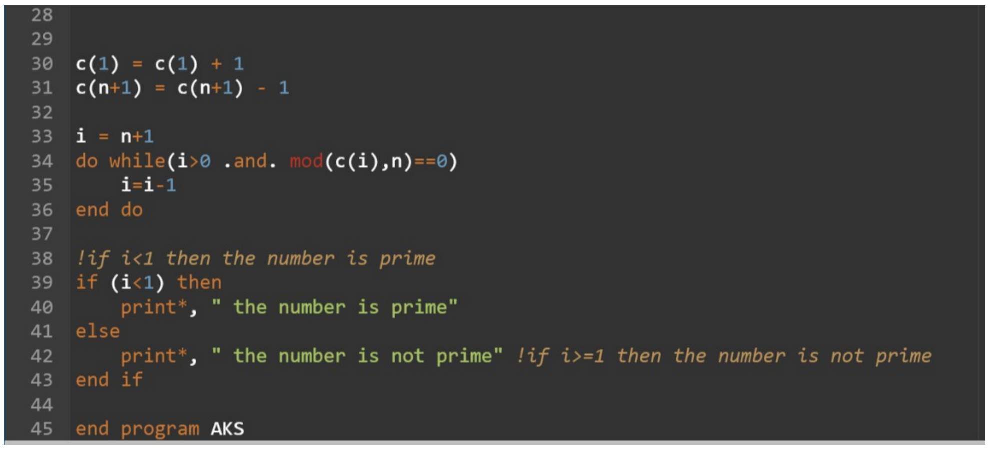
```
1: procedure AKS($n$)                                        ▷ Input is n∈N
2:     System Initialization
3:     Read the value of n
4:     If $\exists$ a,b $>$ 1 $\in$ N such that n = $a^b$, then output composite
5:     Find the minimal r $\in$ N such that $o_r(n) > log^2(n)$
6:     while a=1 to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

1.2.3 Code



```fortran
!AKS PROGRAM
program AKS

!initializing n as integer and c as integer array
integer(kind=16) :: n,c(200)

read*,n

!loop for assigning values of c array to 0
do i=1,200
    c(i) = 0
end do
! ending the loop

!first element to 1
c(1) = 1

!loop traversing from 1 to n
do i=1,n
    c(1+i) = 1

    do j=i,2,-1
        c(j) = c(j-1) - c(j)
    end do

    c(1) = -c(1)
end do
```

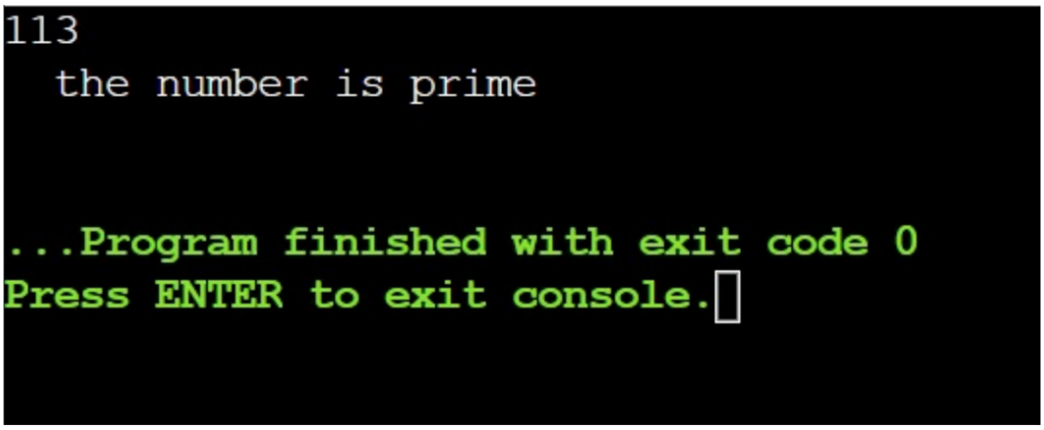
FIGURE 1.3: Part 1 of FORTRAN Code of AKS Primality Test



```fortran


c(1) = c(1) + 1
c(n+1) = c(n+1) - 1

i = n+1
do while(i>0 .and. mod(c(i),n)==0)
    i=i-1
end do

!if i<1 then the number is prime
if (i<1) then
    print*, " the number is prime"
else
    print*, " the number is not prime" !if i>=1 then the number is not prime
end if

end program AKS
```

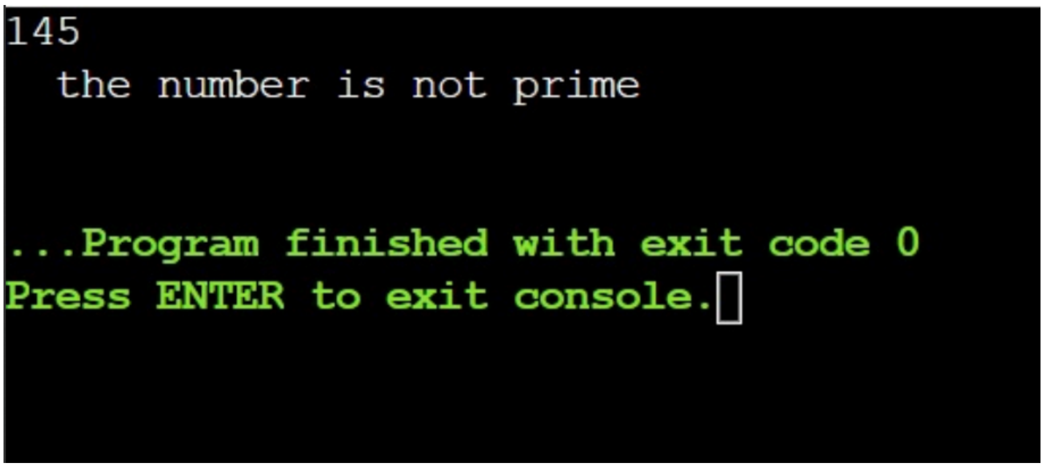
FIGURE 1.4: Part 2 of FORTRAN Code of AKS Primality Test

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 $n^k \neq 1$ mod 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 multiply modulo r each product will have factors less than r. Hence, again by our summary, each multiplication takes $O(logr)$ time. Computing GCD of r numbers where r is bounder by previous step. Computing GCD takes $O(logn)$ time. Therefore, total time complexity for this step is $O(rlogn)$ = $O(log^6 n)$. 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 $O(logn)$ time. Next comes the loop which runs for values of a from 1 to $\lfloor\sqrt{\phi(r)}.log(n)\rfloor$, the time complexity for this step would be $O(\sqrt{r}logn)$. Finally, the step computes $(X+a)^n$ and $X^n + a$ mod $(X^r - 1, n)$. Naively it could take n multiplications to compute $(X+a)^n$ mod (X^r − 1,n). This time dominates all the other ones, so the total runtime of our algorithm is indeed $O(log^{21/2}n)$. [7][8]
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 $O(log^6 n)$. [7][8]

Lemma : Let P(m) be the greatest prime divisor of m. There exists constants c > 0 and n_0 such that, for all x $\geq n_0$ [7][8] :

$$|q|q \in prime, q \leq x, P(q-1) > q^{2/3}| \geq c.\frac{x}{logx} \tag{1.8}$$

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 $o_r(n) > log^2 n$. Using this, the time complexity of the algorithm is brought down to :

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

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) [7][8].

1.3 Trial Division

1.3.1 Introduction

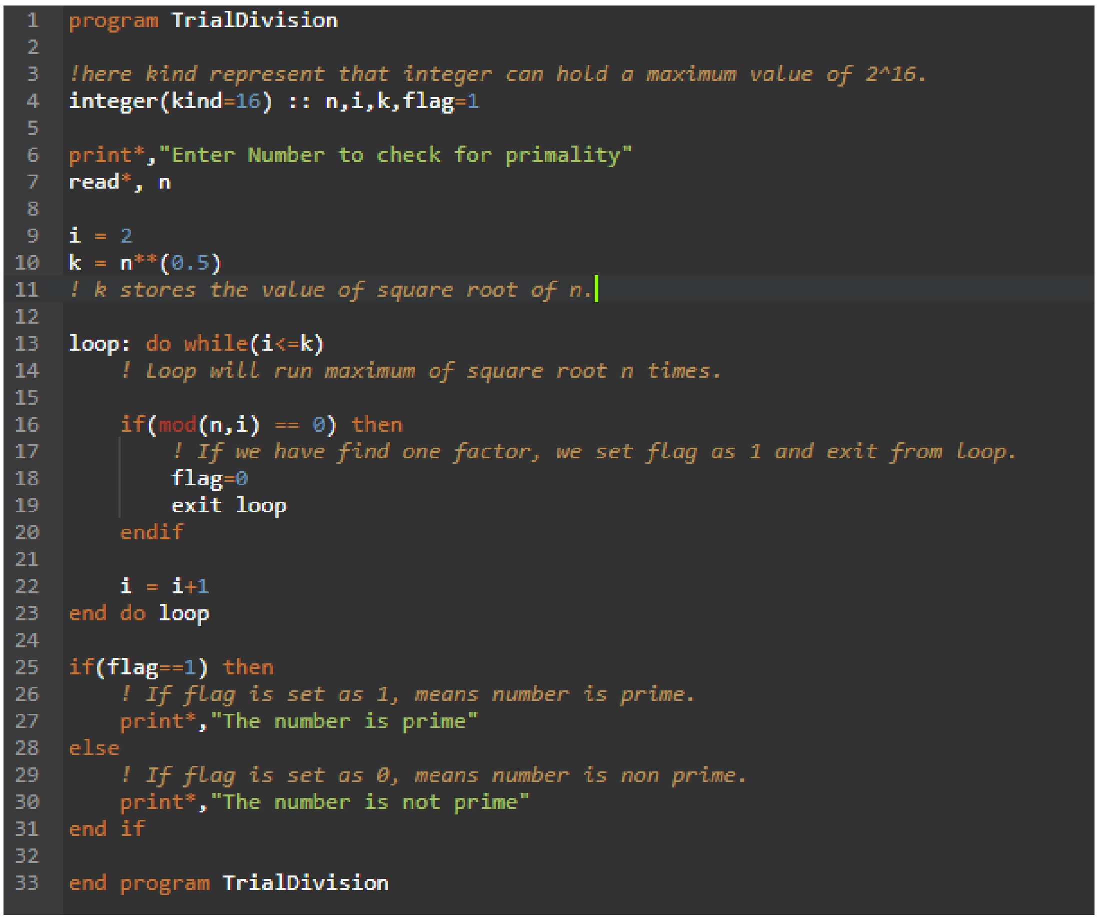
Trial Division is the strenous but easiest primality test. The essential idea behind trial division test is to see if integer **n** can be divided by each number less than **n**. Trial Division was first described by Fibonacci in his book Liber Abaci. The method followed by Trial Division is: Given an 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

```
procedure TRIALDIVISION($n$)                                ▷ Input is n∈N
    System Initialization
    Read the value of n
    while k=2,3,4....$\lfloor\sqrt{n}\rfloor$ do
        if n $\equiv$ 0 (mod k) then
            output Composite
        end if
    end while
    Return n is Prime
end procedure
```

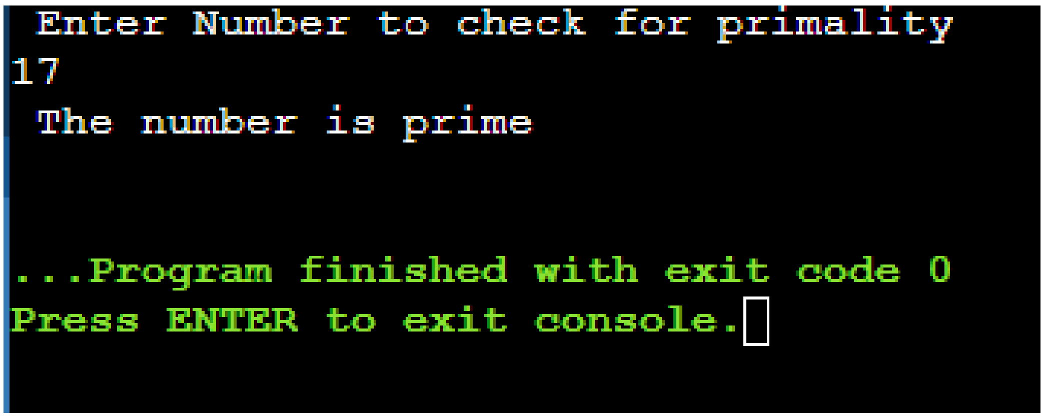
1.3.3 Code



```fortran
program TrialDivision

!here kind represent that integer can hold a maximum value of 2^16.
integer(kind=16) :: n,i,k,flag=1

print*,"Enter Number to check for primality"
read*, n

i = 2
k = n**(0.5)
! k stores the value of square root of n.

loop: do while(i<=k)
    ! Loop will run maximum of square root n times.

    if(mod(n,i) == 0) then
        ! If we have find one factor, we set flag as 1 and exit from loop.
        flag=0
        exit loop
    endif

    i = i+1
end do loop

if(flag==1) then
    ! If flag is set as 1, means number is prime.
    print*,"The number is prime"
else
    ! If flag is set as 0, means number is non prime.
    print*,"The number is not prime"
end if

end program TrialDivision
```

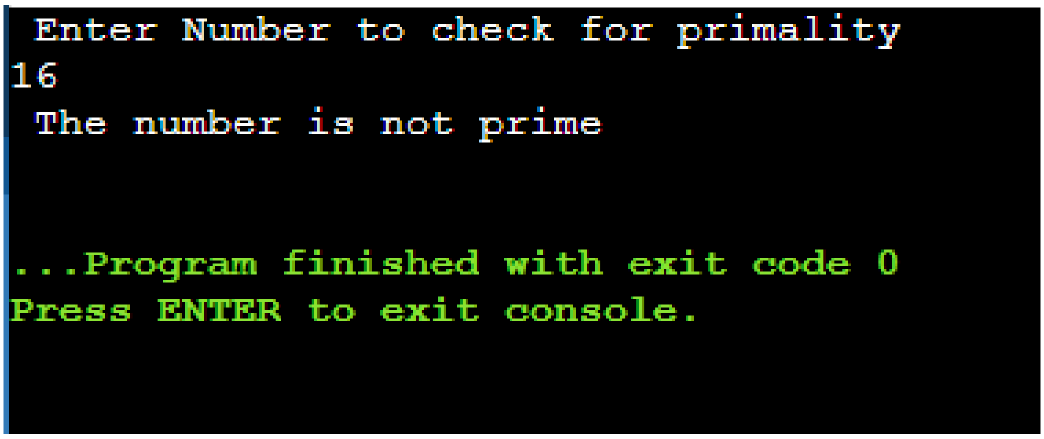
FIGURE 1.7: FORTRAN Code of Trial Division

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,...$\lfloor\sqrt{n}\rfloor$. In the worst case, Trial division requires $O(\sqrt{N})$ divisions to be made.

Trial Division algorithm runs in the time $O(\sqrt{n}.log^2(n))$, 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]

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 18^{th} 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 : Len n $\in$ N. Then n is prime if and only if (n-1)! $\equiv$ -1 (mod n).
Proof: For the forward part, Suppose n is prime. Then every integer in the interval [2,3,4...n-2] is coprime to n and has a unique inverse modulo n. Therefore,

$$\prod_{2\leq j\leq n-2} j \equiv 1(mod(n)) \tag{1.10}$$

and we know that (n-1) $\equiv$ -1 (mod n). Hence,

$$\prod_{2\leq j\leq n-1} j = (n-1)! \equiv -1(mod(n)) \tag{1.11}$$

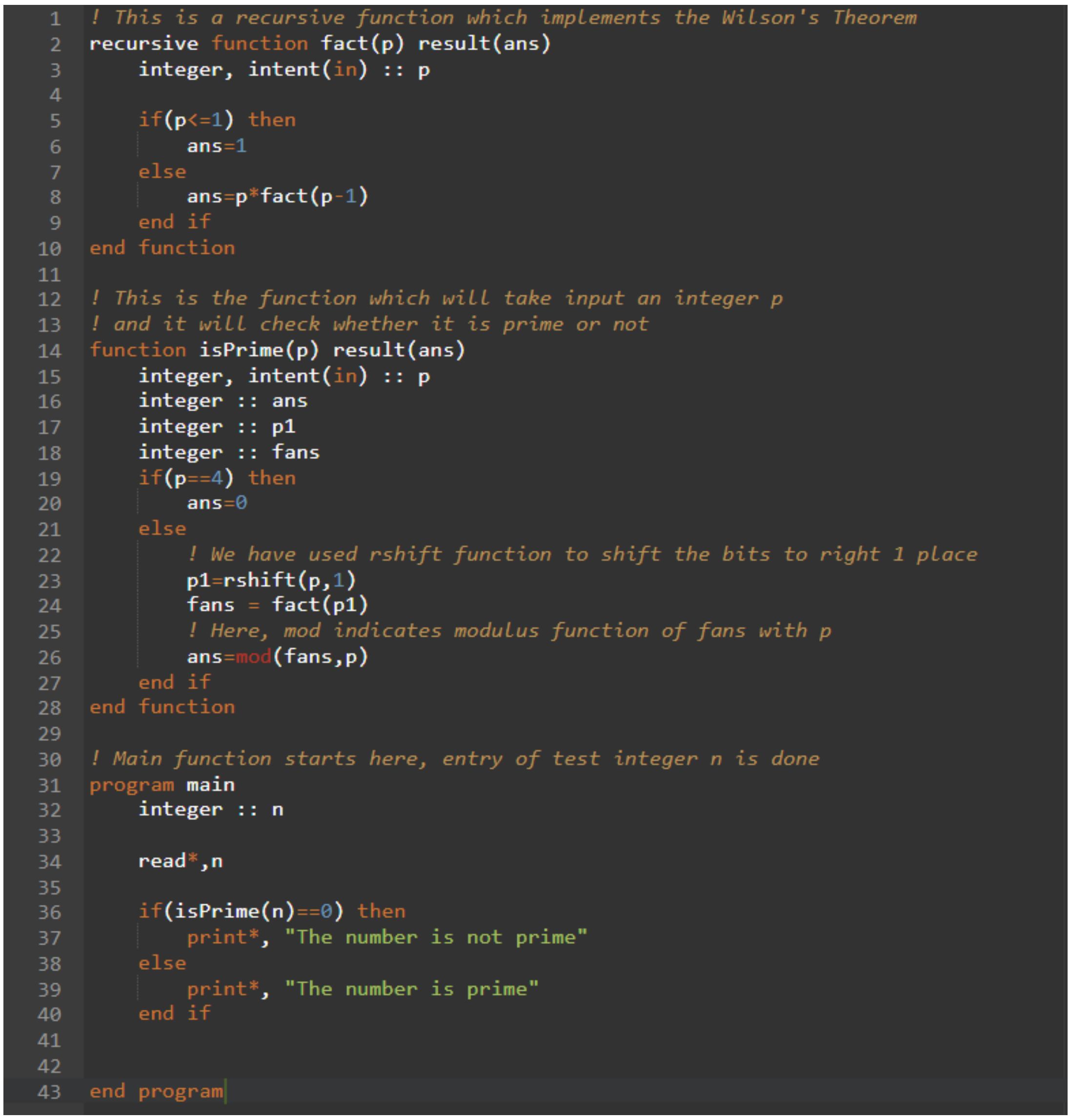
Now let us look at the converse part, suppose that n is composite. Then 1,2,3...n-1 contains all prime factors of n, which implies that (n-1)! $\neq$ -1 (mod n) (because if (n-1)! $\equiv$ -1(mod(n)), then a factor of n, say d will also satisfy this congruence. One can see that (n-1)! $\equiv$ 0(mod(d))).
From the Wilson's characterization of primes, we can determine the primality of an integer n by calculating (n-1)! (mod n). But this computation would require (n-1) multiplications, making it very time consuming. [1]

1.4.2 Algorithm

Algorithm 4 Wilson's Primality Test

```
procedure WILSON($n$)                          ▷ Input is n∈N
    System Initialization
    Read the value of n
    if (n-1)! $\equiv$ -1(mod n) then
        Output Prime
    end if
    otherwise, return Composite
end procedure
```

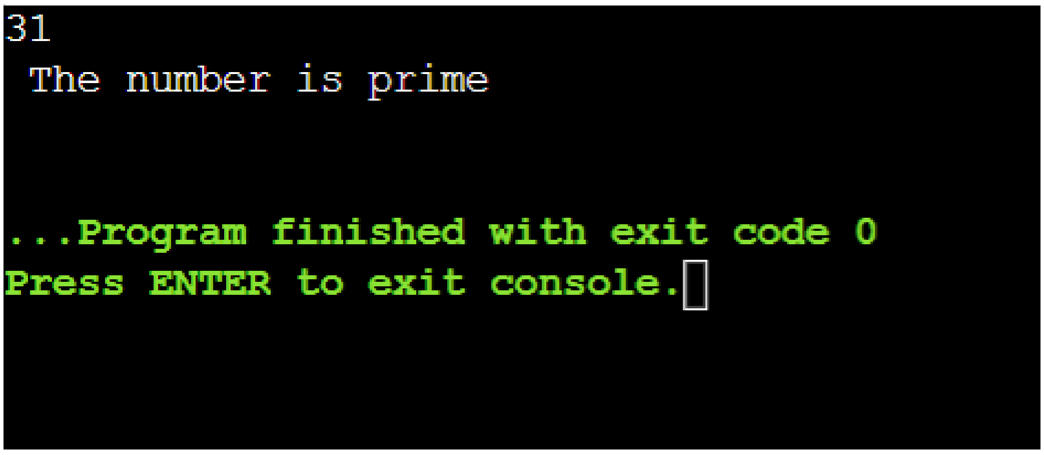
1.4.3 Code



```fortran
! This is a recursive function which implements the Wilson's Theorem
recursive function fact(p) result(ans)
    integer, intent(in) :: p

    if(p<=1) then
        ans=1
    else
        ans=p*fact(p-1)
    end if
end function

! This is the function which will take input an integer p
! and it will check whether it is prime or not
function isPrime(p) result(ans)
    integer, intent(in) :: p
    integer :: ans
    integer :: p1
    integer :: fans
    if(p==4) then
        ans=0
    else
        ! We have used rshift function to shift the bits to right 1 place
        p1=rshift(p,1)
        fans = fact(p1)
        ! Here, mod indicates modulus function of fans with p
        ans=mod(fans,p)
    end if
end function

! Main function starts here, entry of test integer n is done
program main
    integer :: n

    read*,n

    if(isPrime(n)==0) then
        print*, "The number is not prime"
    else
        print*, "The number is prime"
    end if


end program
```

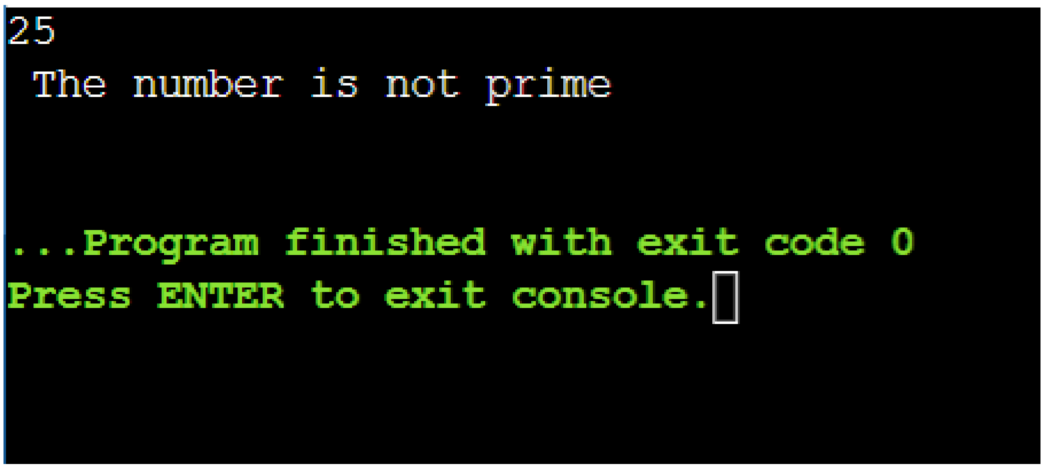
FIGURE 1.10: FORTRAN Code of Wilson's Primality Test

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

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 Lehmer in the 1930s. Now let us talk in brief about Mersenne numbers. [9]
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 17^{th} century. [9]
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≥0 by :

$$s_i = 4, i = 0 \tag{1.12}$$

$$si = s_{i-1}^2, otherwise \tag{1.13}$$

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

$$s_{p-2} \equiv 0(modM_p) \tag{1.14}$$

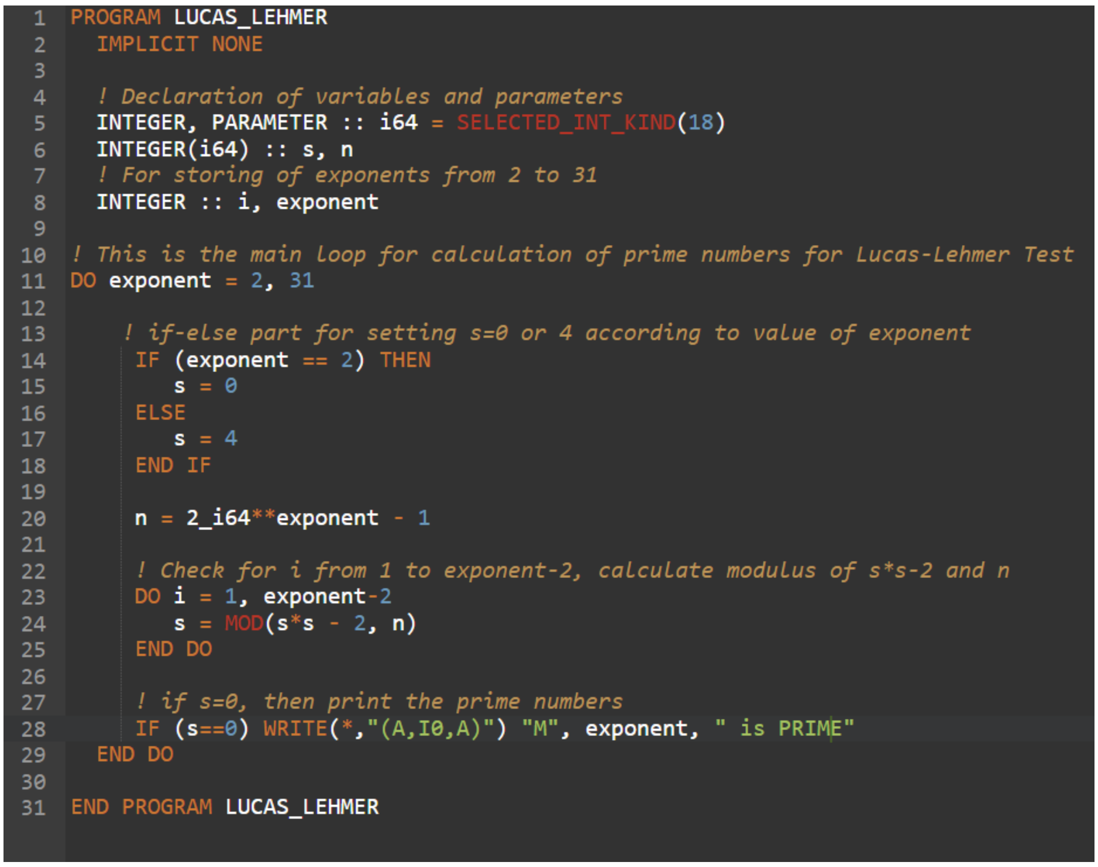
The number s_{p-2} mod M_p is called the Lucas-Lehmer residue of p. Lucas-Lehmer is a deterministic primality test algorihm and it is the last stage in the procedure which is employed by Great Internet Mersenne Prime Search Distributed computing project to find Mersenne primes. [9]

1.5.2 Algorithm

Algorithm 5 Lucas-Lehmer Primality Test

```
procedure LUCAS-LAHMER
    System Initialization
    Declare variable i64
    Declare variables s and n
    Declare variable i and exponent
    while exponent = 2 to 31 do
        if exponent = 2 then
            Let s=0
        else
            Let s=4
        end if
    end while
    Let n = 2_i64**exponent - 1
    while i = 1 to exponent-2 do
        s = (s*s - 2) % n
    end while
    if s=0 then
        return Prime
    end if
end procedure
```

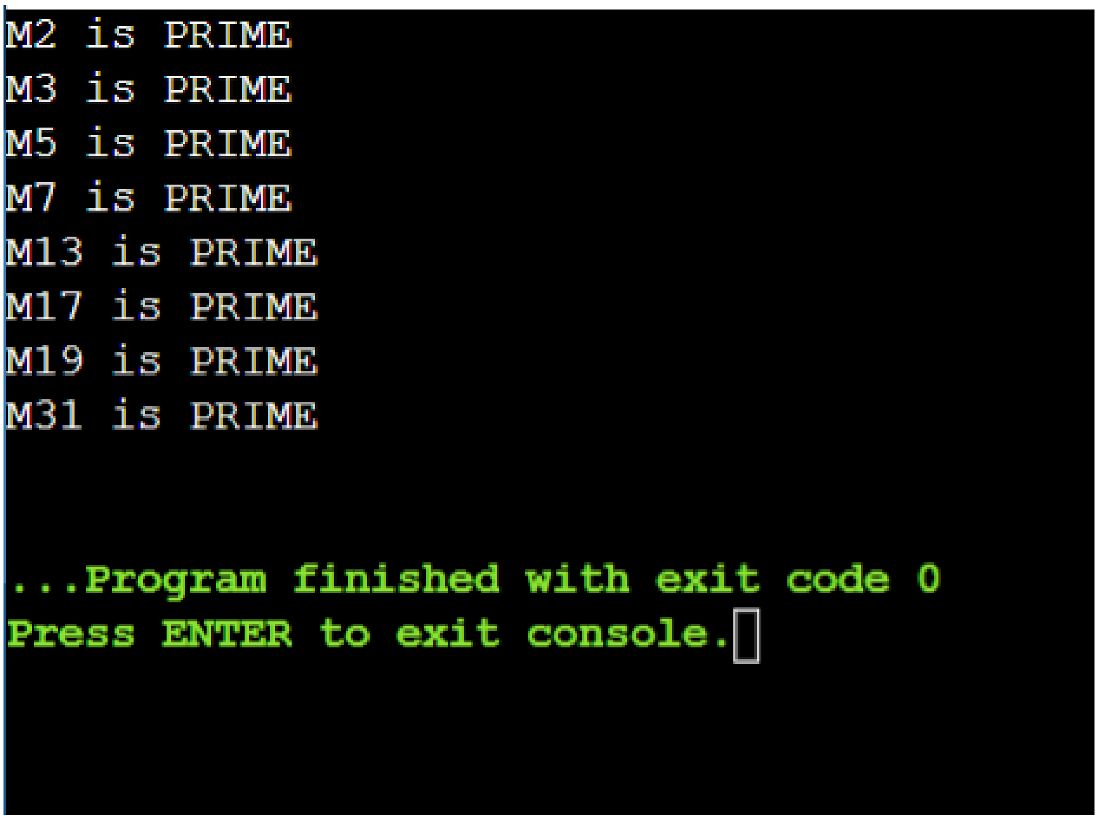
1.5.3 Code



```fortran
PROGRAM LUCAS_LEHMER
  IMPLICIT NONE

  ! Declaration of variables and parameters
  INTEGER, PARAMETER :: i64 = SELECTED_INT_KIND(18)
  INTEGER(i64) :: s, n
  ! For storing of exponents from 2 to 31
  INTEGER :: i, exponent

! This is the main loop for calculation of prime numbers for Lucas-Lehmer Test
DO exponent = 2, 31

    ! if-else part for setting s=0 or 4 according to value of exponent
     IF (exponent == 2) THEN
        s = 0
     ELSE
        s = 4
     END IF

     n = 2_i64**exponent - 1

     ! Check for i from 1 to exponent-2, calculate modulus of s*s-2 and n
     DO i = 1, exponent-2
        s = MOD(s*s - 2, n)
     END DO

     ! if s=0, then print the prime numbers
     IF (s==0) WRITE(*,"(A,I0,A)") "M", exponent, " is PRIME"
  END DO

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 x s, and mod M operation. The mod M operation can be made efficient if we observe that :

$$k \equiv (k(mod2^n)) + \lfloor \frac{k}{2^n} \rfloor (mod2^n - 1) \tag{1.15}$$

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 x 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. [9]
Lucas-Lehmer Primality Test when used with Fast fourier tranform has a complexity of $O(log^2NloglogN)$ where N represents the Mersenne prime number. The complexity can be further reduced down to $O(n^2logn)$. The **space complexity** of Lucas-Lehmer test is O(1) since no extra or auxillary space is used. [9]

1.6 Fermat Primality Test

1.6.1 Introduction

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 mod 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. [10] Fermat's little theorem states that if p is prime and a is not divisible by p, then :

$$a^{p-1} \equiv 1(modp). \tag{1.16}$$

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 (mod 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. [11] Any a such that $a^{n-1} \equiv$ 1 (mod 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 $a^{n-1}! \equiv$ 1 (mod 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 needs to compute x^{n-1} mod 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 $O(logn)$ multiplication operations. [1]

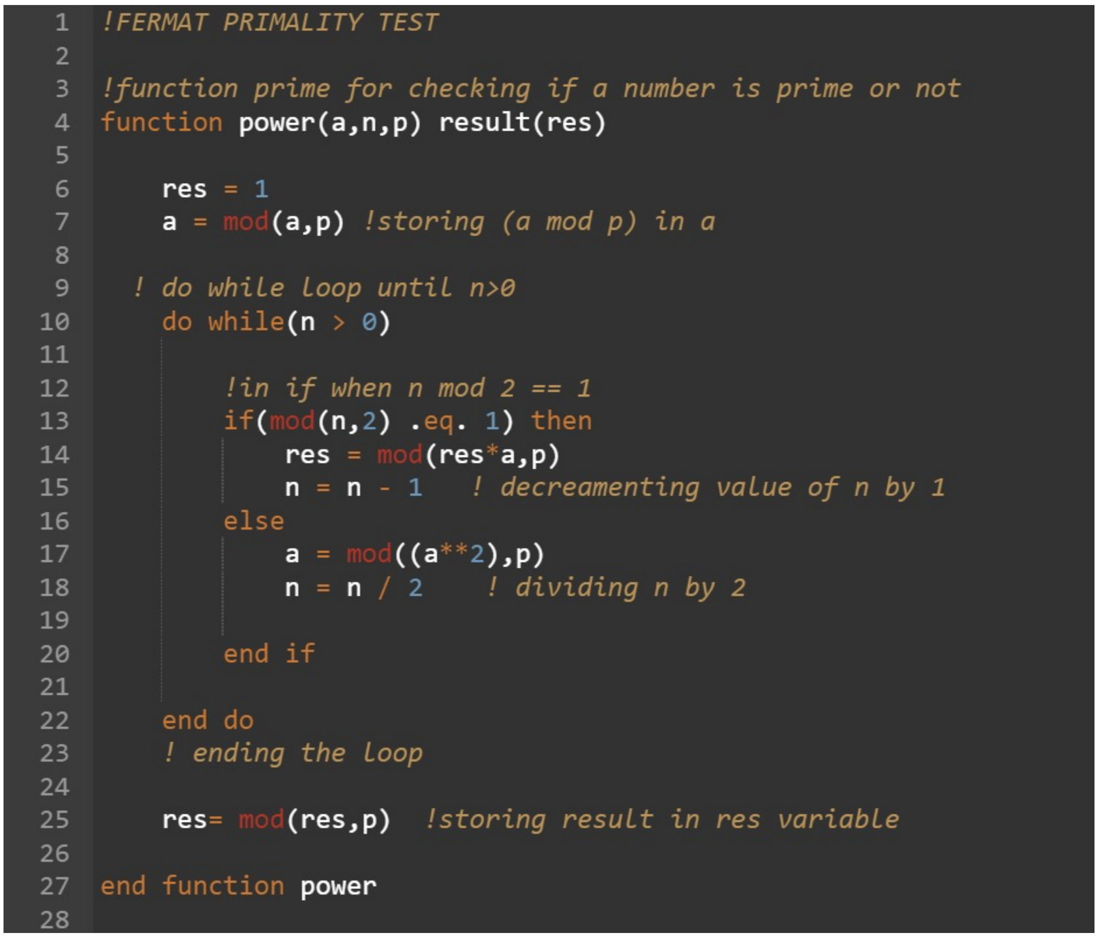
1.6.2 Algorithm

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 probablity of our composite result is correct and for prime, our result is always correct. [1]
Output : It shows whether n is composite or prime.

Algorithm 6 Fermat Primality Test

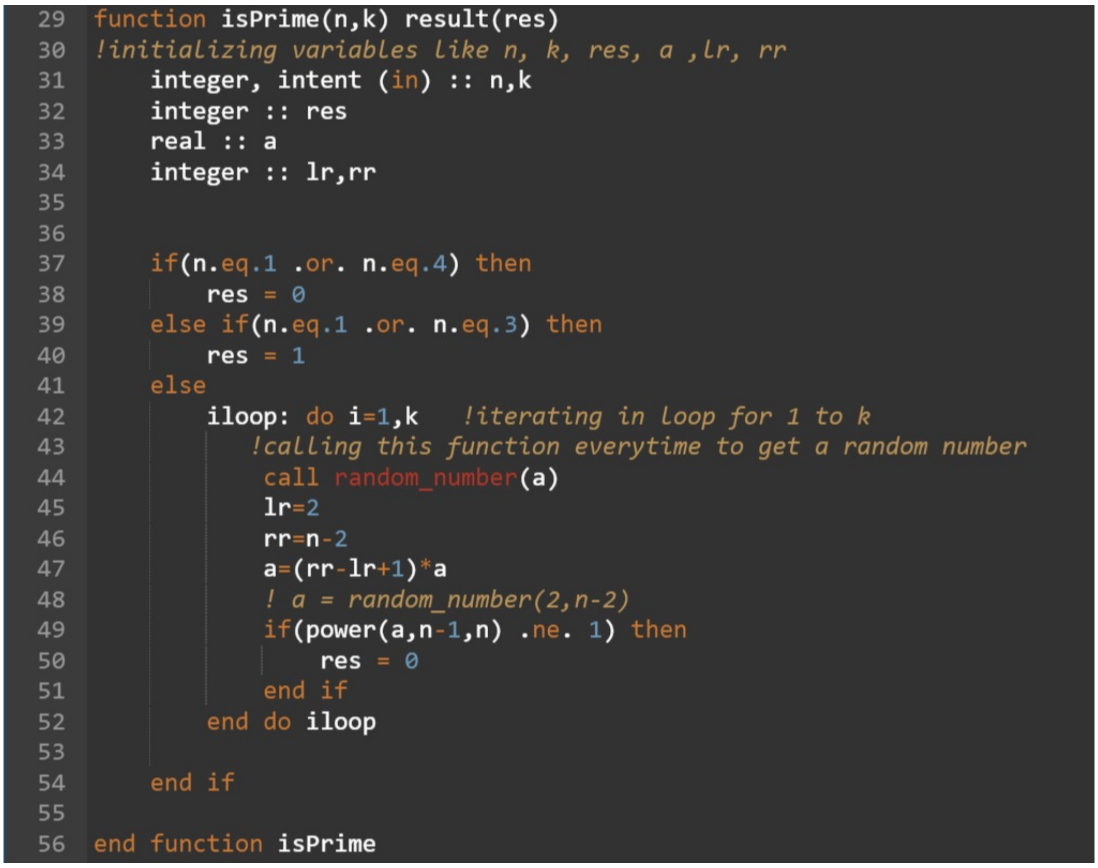
```
1: procedure FERMAT-PRIMALITY($n$)                    ▷ Input is n∈N
2:     System Initialization
3:     Read the value of n
4:     Choose x $\in \{1,2,3,....n-1\}$ uniformly at random
5:     if $x^{n-1} \neq 1$ (mod n) then
6:         return Composite
7:     else
8:         return Probably Prime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

1.6.3 Code



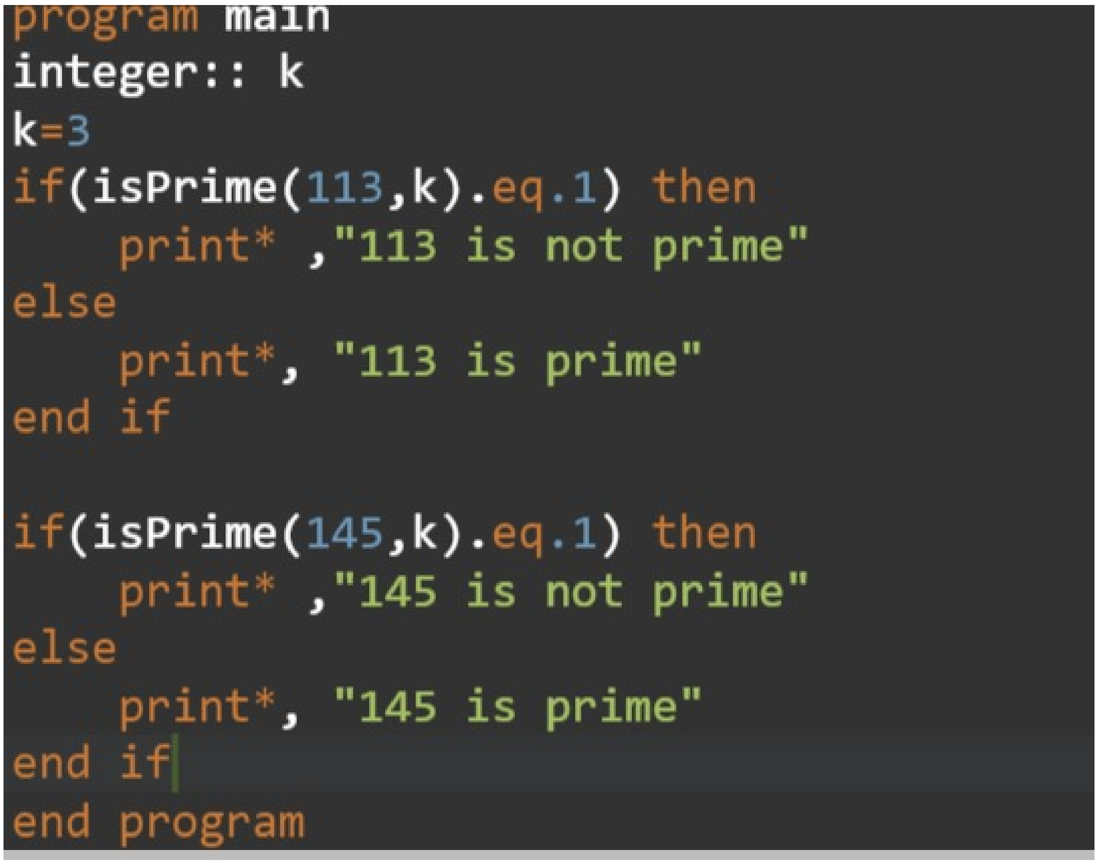
```fortran
!FERMAT PRIMALITY TEST

!function prime for checking if a number is prime or not
function power(a,n,p) result(res)

    res = 1
    a = mod(a,p) !storing (a mod p) in a

  ! do while loop until n>0
    do while(n > 0)

        !in if when n mod 2 == 1
        if(mod(n,2) .eq. 1) then
            res = mod(res*a,p)
            n = n - 1   ! decreamenting value of n by 1
        else
            a = mod((a**2),p)
            n = n / 2     ! dividing n by 2

        end if

    end do
    ! ending the loop

    res= mod(res,p)  !storing result in res variable

end function power

```

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



```fortran
function isPrime(n,k) result(res)
!initializing variables like n, k, res, a ,lr, rr
    integer, intent (in) :: n,k
    integer :: res
    real :: a
    integer :: lr,rr


    if(n.eq.1 .or. n.eq.4) then
        res = 0
    else if(n.eq.1 .or. n.eq.3) then
        res = 1
    else
        iloop: do i=1,k   !iterating in loop for 1 to k
           !calling this function everytime to get a random number
            call random_number(a)
            lr=2
            rr=n-2
            a=(rr-lr+1)*a
            ! a = random_number(2,n-2)
            if(power(a,n-1,n) .ne. 1) then
                res = 0
            end if
        end do iloop

    end if

end function isPrime
```

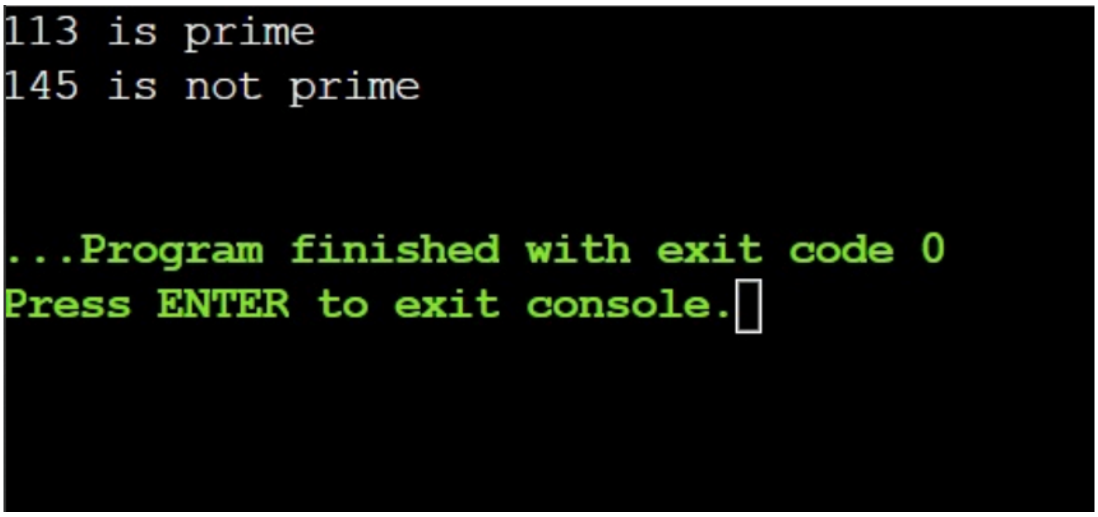
FIGURE 1.16: Part 2 of FORTRAN Code of Fermat Primality Test



```
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

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*(Time complexity of power function)).
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.

$$T(n) = T(n/2) + c1 \tag{1.17}$$

The above equation holds if n is even.

$$T(n) = T(n-1) + c2 \tag{1.18}$$

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

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. [12]

1.7 Miller Rabin Primality Test

1.7.1 Introduction

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. [1]
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 $a^d \equiv 1$ (mod 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 [1] :

Lemma (Fake Square root lemma) : If x,n are positive integers such that $x^2 \equiv 1$ (mod n) but $x \neq \pm 1$ (mod 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. [1]

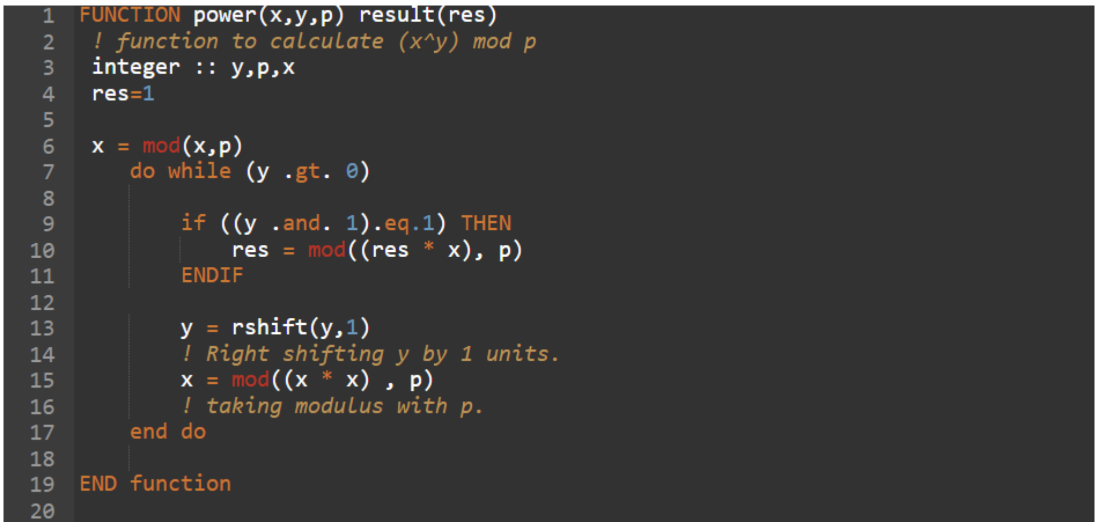
Idea behind Miller-Rabin Test : The idea of the test is to pick a random number x in $\{$1,2,....,n-1$\}$ and use it to try finding either a Fermat witness or a fake square root of 1 mod n. [1]

1.7.2 Algorithm

Algorithm 7 Miller-Rabin [1]

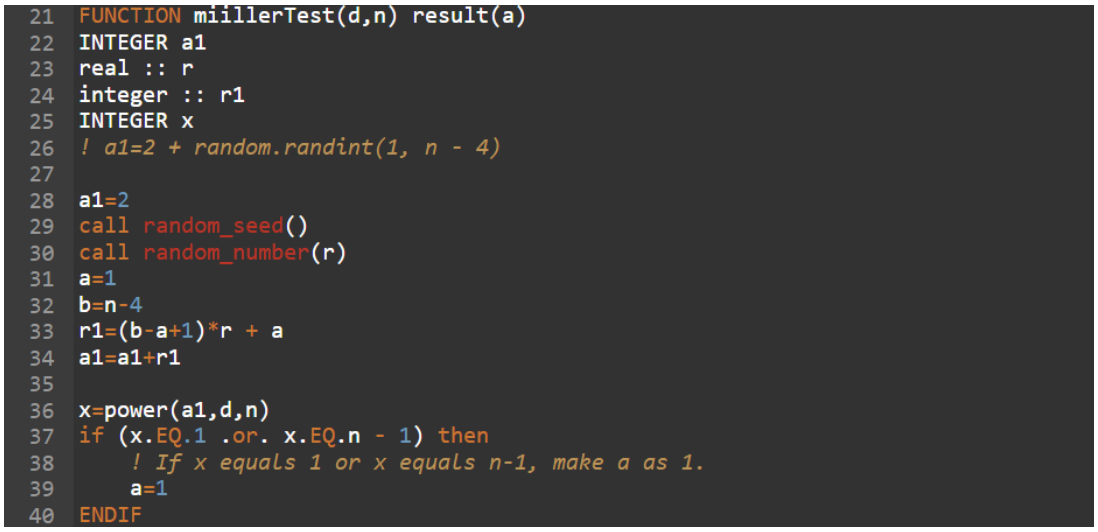
```
procedure MILLER-RABIN($n$)                                   ▷ Input is n∈N
    System Initialization
    Read the value of n
    if n $>$ 2 and n is even then
        return Composite
    end if
    Choose x $\in \{$1,2,3,....,n-1$\}$ uniformly random.
    if $x^{n-1} \neq 1$ (mod n) then
        return Composite
    else
        Factor n-1 = $2^s.t$, where t is odd
        Compute $u_i$ = $x^{2^i.t}$(mod(n)) , where 0 $\leq$ i $<$ s.
        If there is an i such that $u_i$ = 1 and $u_{i-1} \neq \pm 1$, then return Composite.
    end if
    Return Probably Prime
end procedure
```

1.7.3 Code



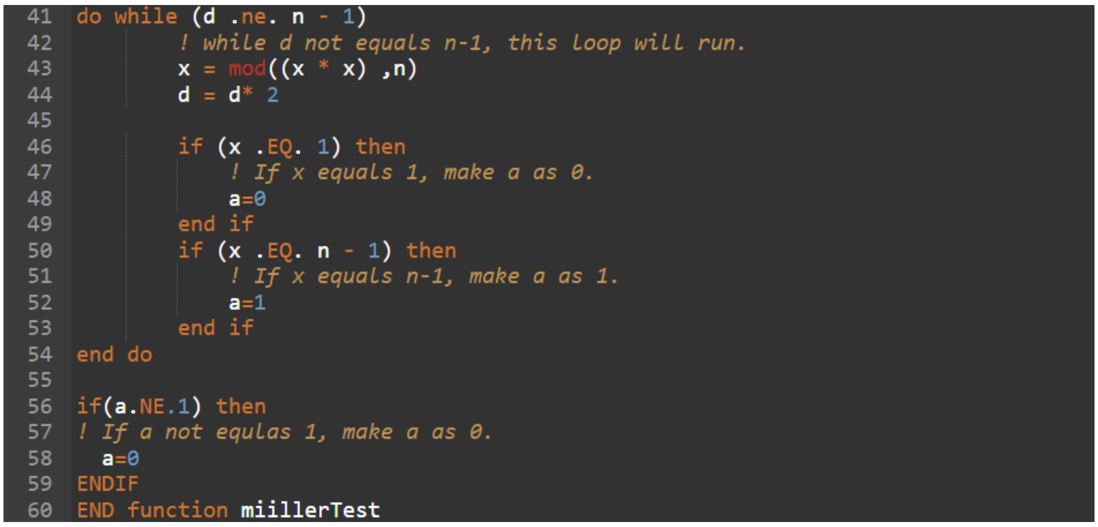
```fortran
FUNCTION power(x,y,p) result(res)
 ! function to calculate (x^y) mod p
 integer :: y,p,x
 res=1

 x = mod(x,p)
    do while (y .gt. 0)

        if ((y .and. 1).eq.1) THEN
            res = mod((res * x), p)
        ENDIF

        y = rshift(y,1)
        ! Right shifting y by 1 units.
        x = mod((x * x) , p)
        ! taking modulus with p.
    end do

END function

```

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



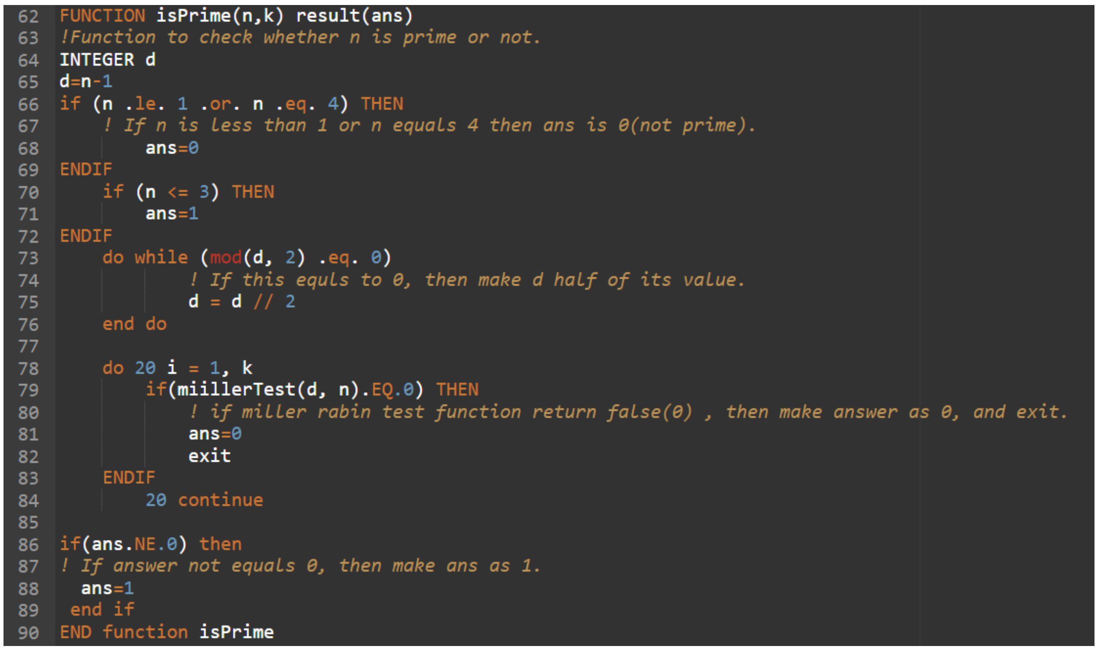
```fortran
FUNCTION miillerTest(d,n) result(a)
INTEGER a1
real :: r
integer :: r1
INTEGER x
! a1=2 + random.randint(1, n - 4)

a1=2
call random_seed()
call random_number(r)
a=1
b=n-4
r1=(b-a+1)*r + a
a1=a1+r1

x=power(a1,d,n)
if (x.EQ.1 .or. x.EQ.n - 1) then
    ! If x equals 1 or x equals n-1, make a as 1.
    a=1
ENDIF
```

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



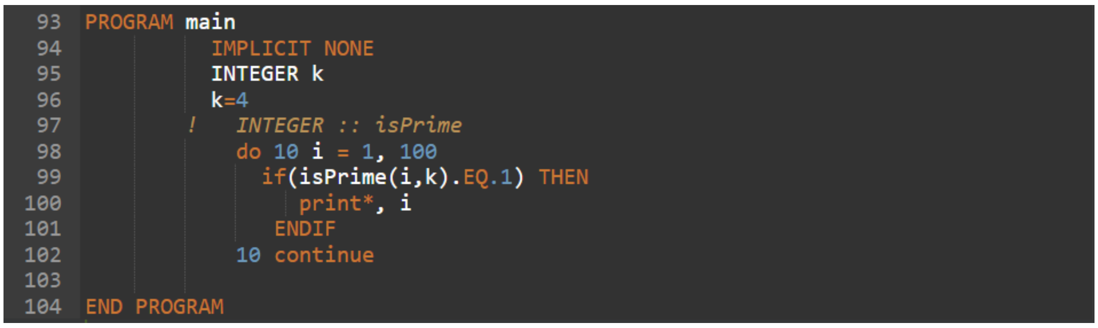
```
do while (d .ne. n - 1)
        ! while d not equals n-1, this loop will run.
        x = mod((x * x) ,n)
        d = d* 2

        if (x .EQ. 1) then
            ! If x equals 1, make a as 0.
            a=0
        end if
        if (x .EQ. n - 1) then
            ! If x equals n-1, make a as 1.
            a=1
        end if
end do

if(a.NE.1) then
! If a not equlas 1, make a as 0.
  a=0
ENDIF
END function miillerTest
```

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



```
FUNCTION isPrime(n,k) result(ans)
!Function to check whether n is prime or not.
INTEGER d
d=n-1
if (n .le. 1 .or. n .eq. 4) THEN
    ! If n is less than 1 or n equals 4 then ans is 0(not prime).
        ans=0
ENDIF
    if (n <= 3) THEN
        ans=1
ENDIF
    do while (mod(d, 2) .eq. 0)
            ! If this equls to 0, then make d half of its value.
            d = d // 2
    end do

    do 20 i = 1, k
        if(miillerTest(d, n).EQ.0) THEN
            ! if miller rabin test function return false(0) , then make answer as 0, and exit.
            ans=0
            exit
    ENDIF
        20 continue

if(ans.NE.0) then
! If answer not equals 0, then make ans as 1.
  ans=1
 end if
END function isPrime
```

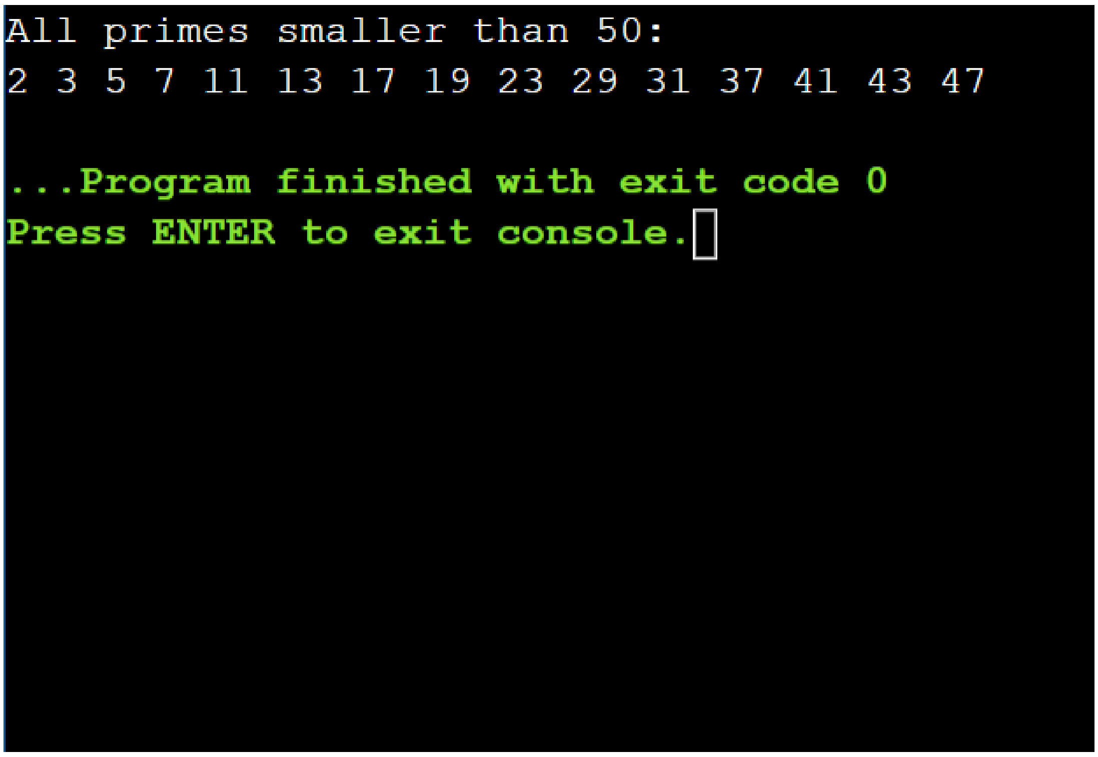
FIGURE 1.22: Part 4 of FORTRAN Code of Miller-Rabin Primality Test



```
PROGRAM main
          IMPLICIT NONE
          INTEGER k
          k=4
        !   INTEGER :: isPrime
            do 10 i = 1, 100
              if(isPrime(i,k).EQ.1) THEN
                print*, i
               ENDIF
            10 continue

END PROGRAM
```

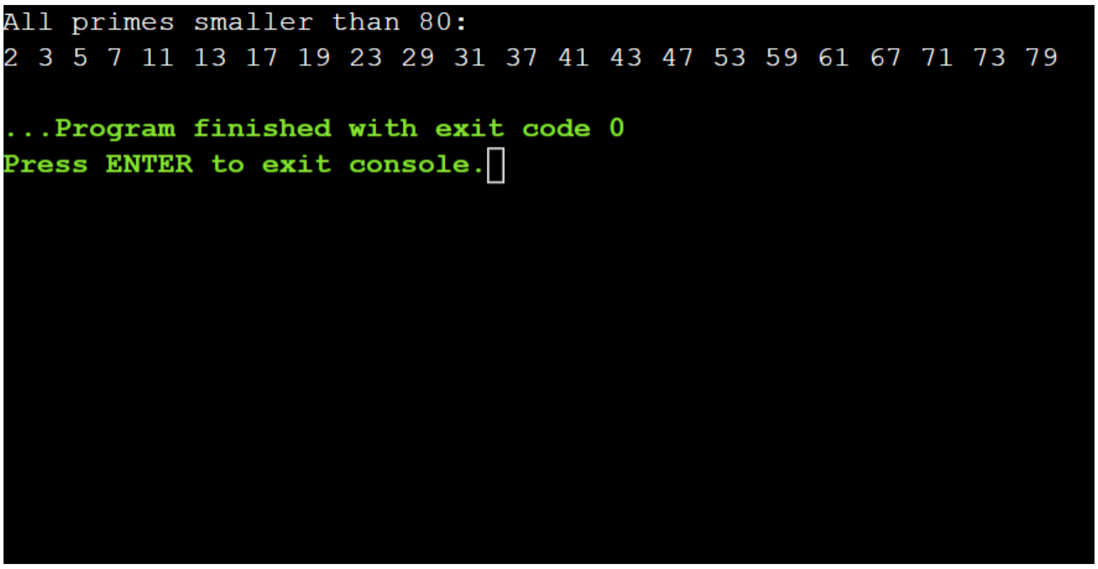
FIGURE 1.23: Part 5 of FORTRAN Code of Miller-Rabin Primality Test

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.
```

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 (mod 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 (mod n), the remaining numbers x^{2t}, x^{4t}, ..., x^{2*t} (mod n) may be obtained by s $\leq log_2(n)$ iterations of repeated squaring mod n, which again provide O(log n) mod-n miltiplication 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

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 is congruent to a (mod p). [13] Lucas primality test is based on the Lucas theorem. The Lucas Theorem can be stated as follows :

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

1. $a^{(n-1)}$ congruent to 1 (mod n) and
2. $a^{((n-1)/p)}$ is not congruent to 1 (mod n), then n is prime.

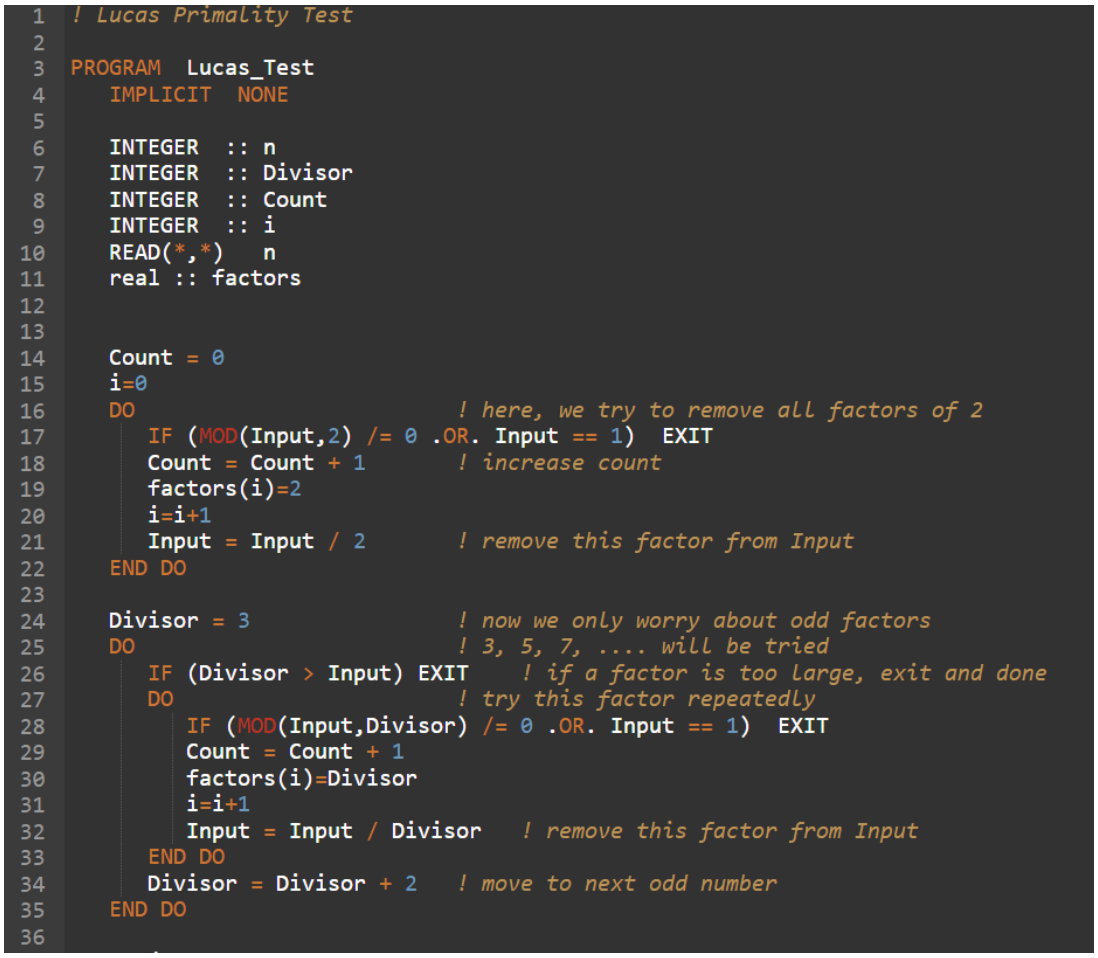
Proof : Suppose n satisfies the conditions of the theorem. To show that n is prime, it is enough to show that Q(n) = n - 1. Since in general Q(n) $<$ n - 1, to show equality we will show that under the above conditions n - 1 divides Q(n) (which means Q(n) $\geq$ n-1 $\Rightarrow$ Q(n) = n-1). Suppose not. Then there exists a prime p such that p^r divides n - 1 but pr 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]

1.8.2 Algorithm

Algorithm 8 Lucas [1]

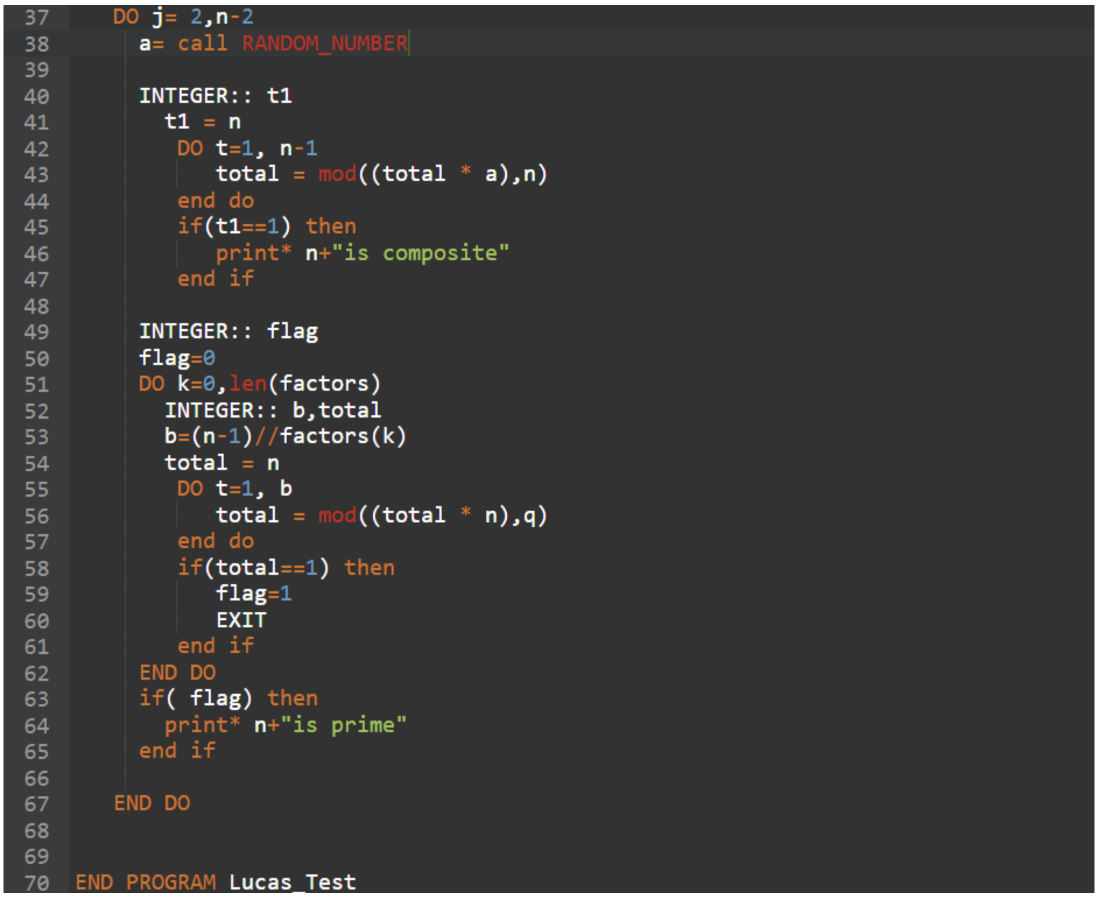
```
1: procedure LUCAS($n$)                                   ▷ Input is n∈N and n $\geq$ 2
2:     System Initialization
3:     Read the value of n
4:     Factor n-1 in to prime factors
5:     while i=2 to n-1 do
6:         Choose an a such that gcd(a, n) = 1
7:         if $a^{n-1} \cong$ 1 (mod n) and $a^{\frac{n-1}{p}}$ != 1 (mod n) then      ▷ ∀ prime factor p
8:             Return Prime
9:         end if
10:    end while
11:    Return Probably Composite
12: end procedure
```

1.8.3 Code



```fortran
! Lucas Primality Test

PROGRAM  Lucas_Test
   IMPLICIT  NONE

   INTEGER  :: n
   INTEGER  :: Divisor
   INTEGER  :: Count
   INTEGER  :: i
   READ(*,*)   n
   real :: factors


   Count = 0
   i=0
   DO                           ! here, we try to remove all factors of 2
      IF (MOD(Input,2) /= 0 .OR. Input == 1)  EXIT
      Count = Count + 1         ! increase count
      factors(i)=2
      i=i+1
      Input = Input / 2         ! remove this factor from Input
   END DO

   Divisor = 3                  ! now we only worry about odd factors
   DO                           ! 3, 5, 7, .... will be tried
      IF (Divisor > Input) EXIT     ! if a factor is too large, exit and done
      DO                        ! try this factor repeatedly
         IF (MOD(Input,Divisor) /= 0 .OR. Input == 1)  EXIT
         Count = Count + 1
         factors(i)=Divisor
         i=i+1
         Input = Input / Divisor    ! remove this factor from Input
      END DO
      Divisor = Divisor + 2     ! move to next odd number
   END DO

```

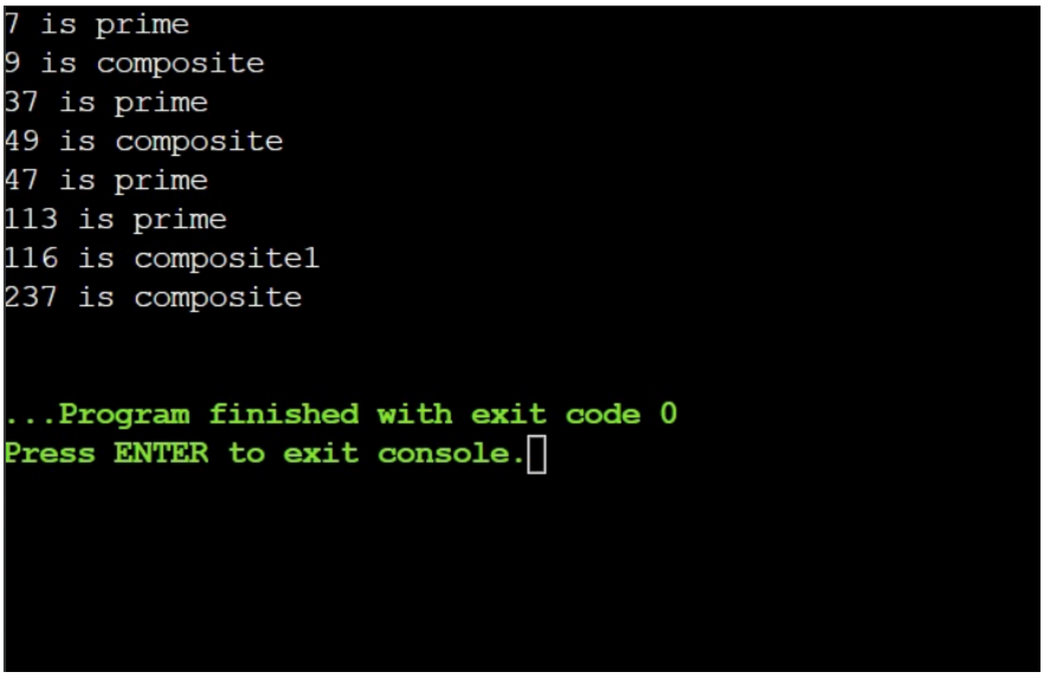
FIGURE 1.26: Part 1 of FORTRAN Code of Lucas Primality Test



```
   DO j= 2,n-2
     a= call RANDOM_NUMBER

     INTEGER:: t1
       t1 = n
        DO t=1, n-1
           total = mod((total * a),n)
        end do
        if(t1==1) then
           print* n+"is composite"
        end if

     INTEGER:: flag
     flag=0
     DO k=0,len(factors)
       INTEGER:: b,total
       b=(n-1)//factors(k)
       total = n
        DO t=1, b
           total = mod((total * n),q)
        end do
        if(total==1) then
           flag=1
           EXIT
        end if
     END DO
     if( flag) then
       print* n+"is prime"
     end if

   END DO


END PROGRAM Lucas_Test
```

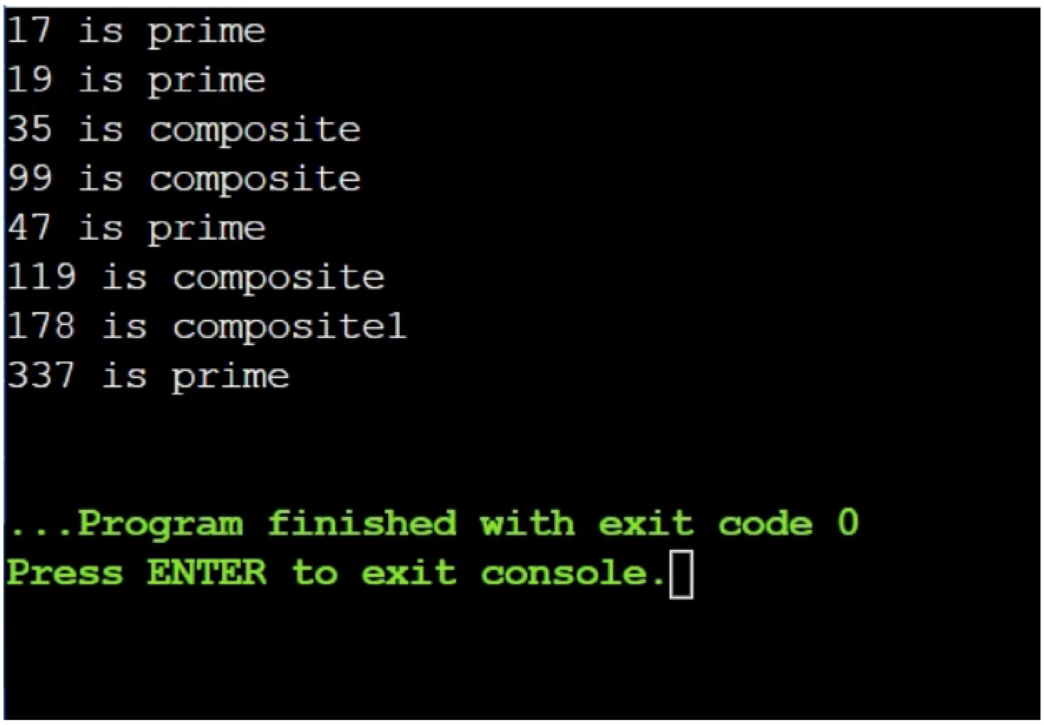
FIGURE 1.27: Part 2 of FORTRAN Code of Lucas Primality Test

1.8.4 Results



```
7 is prime
9 is composite
37 is prime
49 is composite
47 is prime
113 is prime
116 is composite1
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 composite1
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. 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.

$$T(n) = T(n/2) + c1 \tag{1.19}$$

The above equation holds if n is even.

$$T(n) = T(n-1) + c2 \tag{1.20}$$

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 = n 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. [12]

Chapter 2

Prime Factorization Algorithms

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

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 :

$$N = a^2 - b^2 \tag{2.1}$$

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. [1]
The basic idea behind the Fermat factorization algorithm is as follows :

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

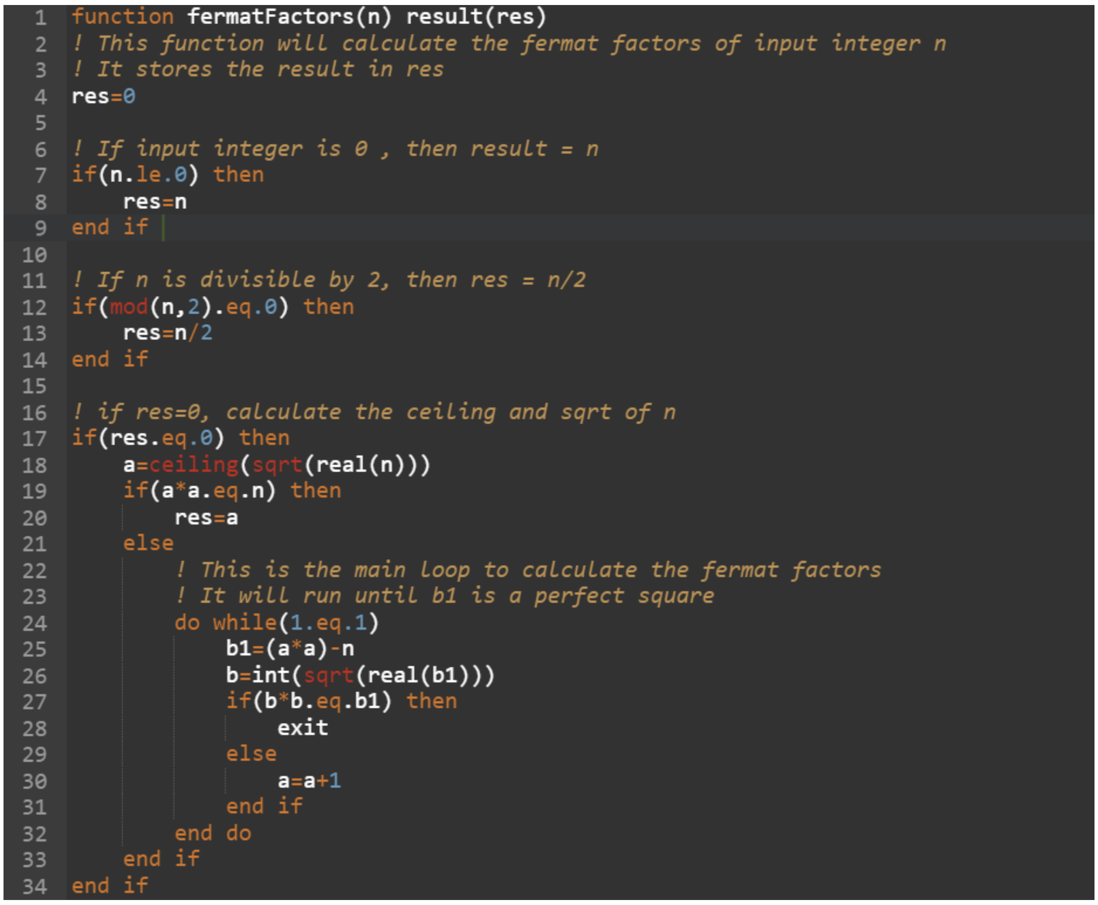
2. gcd(t+s, n) is a non-trivial factor of n

2.1.2 Algorithm

Algorithm 9 Fermat's Factorization

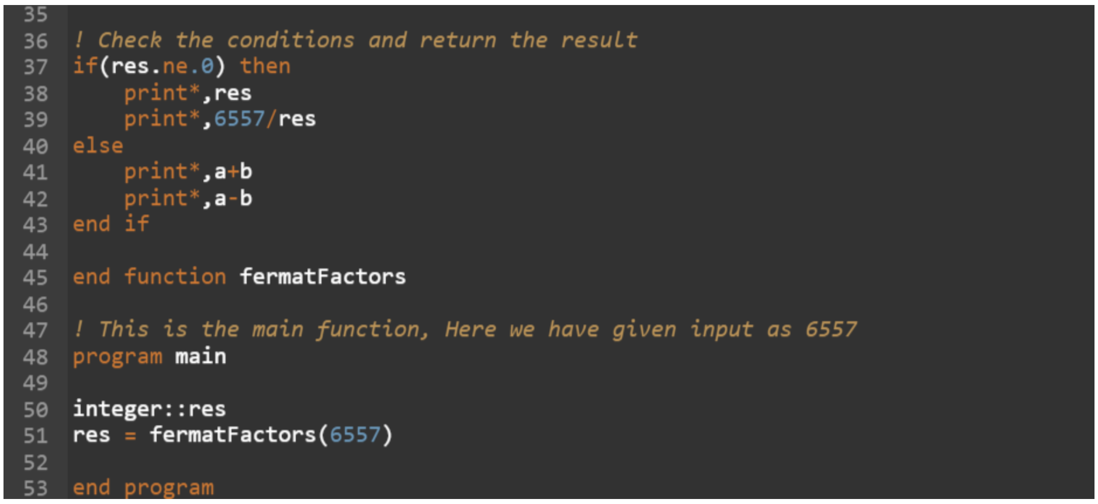
```
procedure FERMAT-FACTOR($N$)                  ▷ Input is n∈N and n is odd
    System Initialization
    Read the value of n
    a ← ceiling(sqrt(N))
    b2 ← a*a - N
    while b2 is a square do
        a ← a + 1
        b2 ← a*a - N
    end while
    return a - sqrt(b2)
end procedure
```

2.1.3 Code



```fortran
function fermatFactors(n) result(res)
! This function will calculate the fermat factors of input integer n
! It stores the result in res
res=0

! If input integer is 0 , then result = n
if(n.le.0) then
    res=n
end if

! If n is divisible by 2, then res = n/2
if(mod(n,2).eq.0) then
    res=n/2
end if

! if res=0, calculate the ceiling and sqrt of n
if(res.eq.0) then
    a=ceiling(sqrt(real(n)))
    if(a*a.eq.n) then
        res=a
    else
        ! This is the main Loop to calculate the fermat factors
        ! It will run until b1 is a perfect square
        do while(1.eq.1)
            b1=(a*a)-n
            b=int(sqrt(real(b1)))
            if(b*b.eq.b1) then
                exit
            else
                a=a+1
            end if
        end do
    end if
end if
```

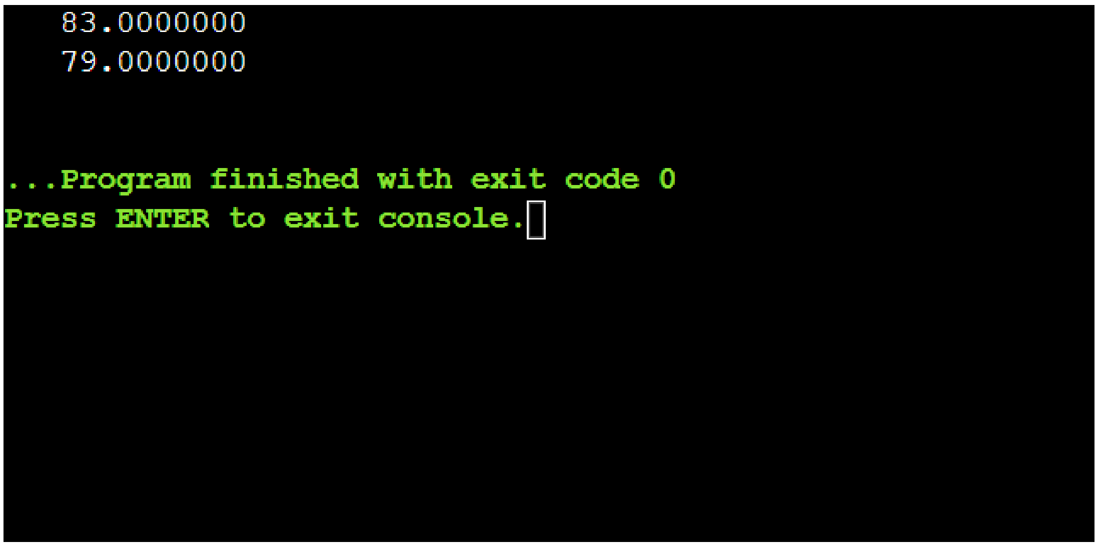
FIGURE 2.1: Part 1 of FORTRAN Code of Fermat Factorization



```fortran

! Check the conditions and return the result
if(res.ne.0) then
    print*,res
    print*,6557/res
else
    print*,a+b
    print*,a-b
end if

end function fermatFactors

! This is the main function, Here we have given input as 6557
program main

integer::res
res = fermatFactors(6557)

end program
```

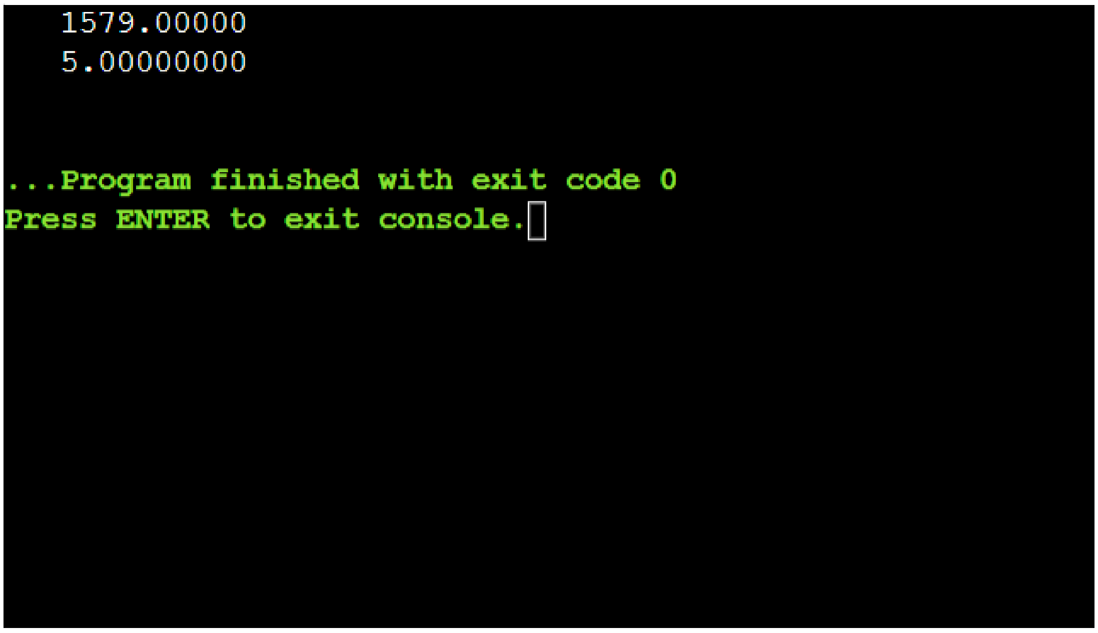
FIGURE 2.2: Part 2 of FORTRAN Code of Fermat Factorization

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
   5.00000000


...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

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 :

$$lim_{n\to\infty}\frac{n!}{\sqrt{2n\pi}n^{n}e^{-n}} = 1 \tag{2.2}$$

We can say that log(n!) is approximately equal to nlog(n)-n.
<u>Idea of Proof</u> : The striling 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 = nlog(n) - n + 1.
Now Let us see a Lemma which talks about the total number of non-negative integers with some binomial coefficients. [1]

<u>Lemma</u> : 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 ($\frac{[u]+N}{N}$).
<u>Idea of Proof</u> : Each N-tuple solution α_j correspond to the following choice of N integers β_j from among 1,2,....[u]+N. Let β_1 = α_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]

<u>Theorem</u> : The overall time complexity of the Fermat factor base algorithm is $O(e^{c\sqrt{rlog(r)}})$ for some constant c, where n is an r-bit integer. [1]

After careful analysis of the above theorems and lemma, we come to the conclusion that time complexity of Fermat factorization algorithm is $O(e^{2ks})$ = $O(e^{\sqrt{2k}\sqrt{rlog(r)}})$. Thus after replacing the constant $\sqrt{2k}$ by C, we finally obtain $O(e^{c\sqrt{rlog(r)}})$ 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. [1]

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.q 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)mod n, and polynomial is used to generate a pseudorandom sequence. [1] [14]
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 -

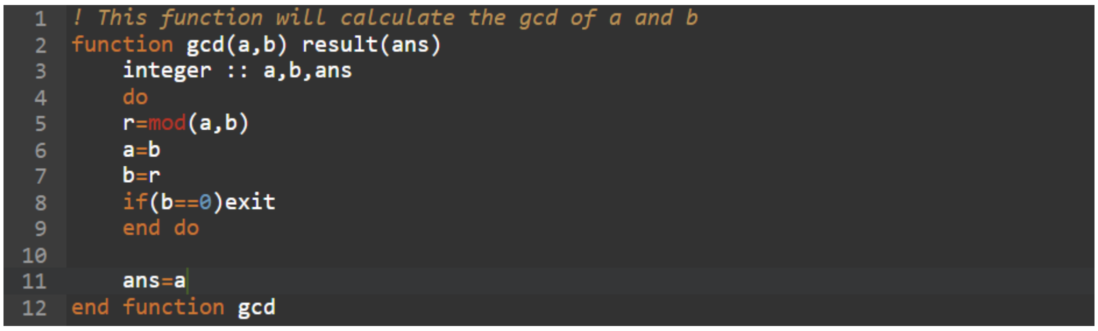
1. Choose an easily evaluated map from Z/nZ to itself, which is a fairly simple polynomial with integer coefficients such as x^2 + a.
2. We will choose some particular value x = x_o.
3. We will compute the successive iterates of f : x_1 = f(x_o), x_2 = f(f(x_o)), x_3 = f(f(f(x_o))). We define x_{j+1} = f(x_j) where j=0,1,2...
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 division of n.
5. We will compute the gcd($x_j - x_k$, n), which is equal to some proper divisor of n. [1]

2.2.2 Algorithm

Algorithm 10 Pollard Rho Factorization

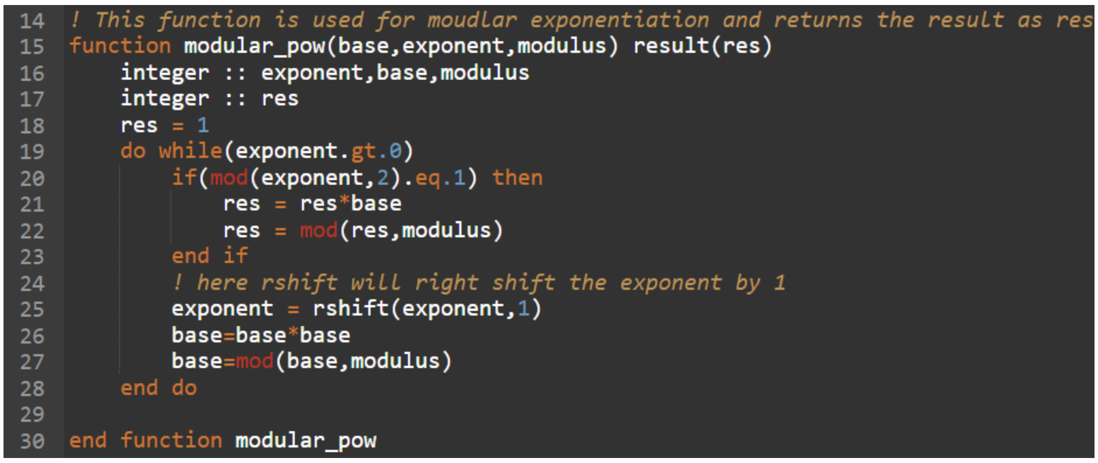
```
procedure POLLARD-FACTOR($N$)    ▷ Input is n∈N which needs to be factorized
    System Initialization
    Read the value of n
    x ← 2
    y ← 2
    d ← 1
    while d = 1 do
        x ← g(x)
        y ← g(g(y))
        d ← gcd($|x - y|$, n)
    end while
    if d = n then
        return failure
    else
        return d
    end if
end procedure
```

2.2.3 Code



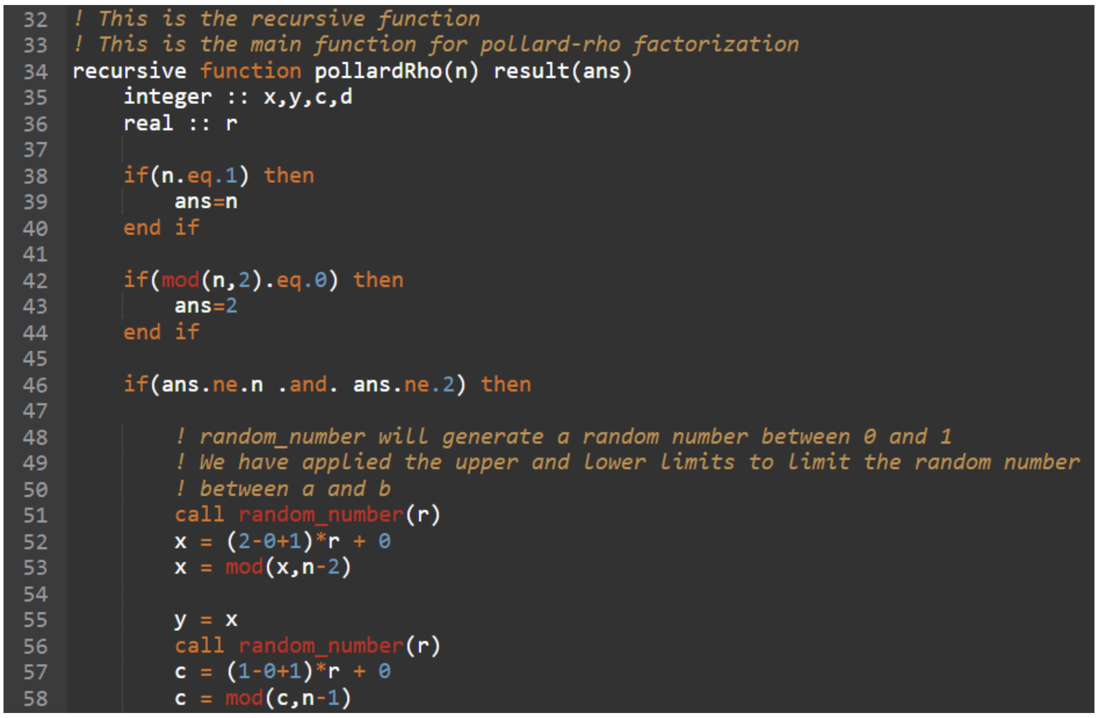
```fortran
! This function will calculate the gcd of a and b
function gcd(a,b) result(ans)
    integer :: a,b,ans
    do
    r=mod(a,b)
    a=b
    b=r
    if(b==0)exit
    end do

    ans=a
end function gcd
```

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



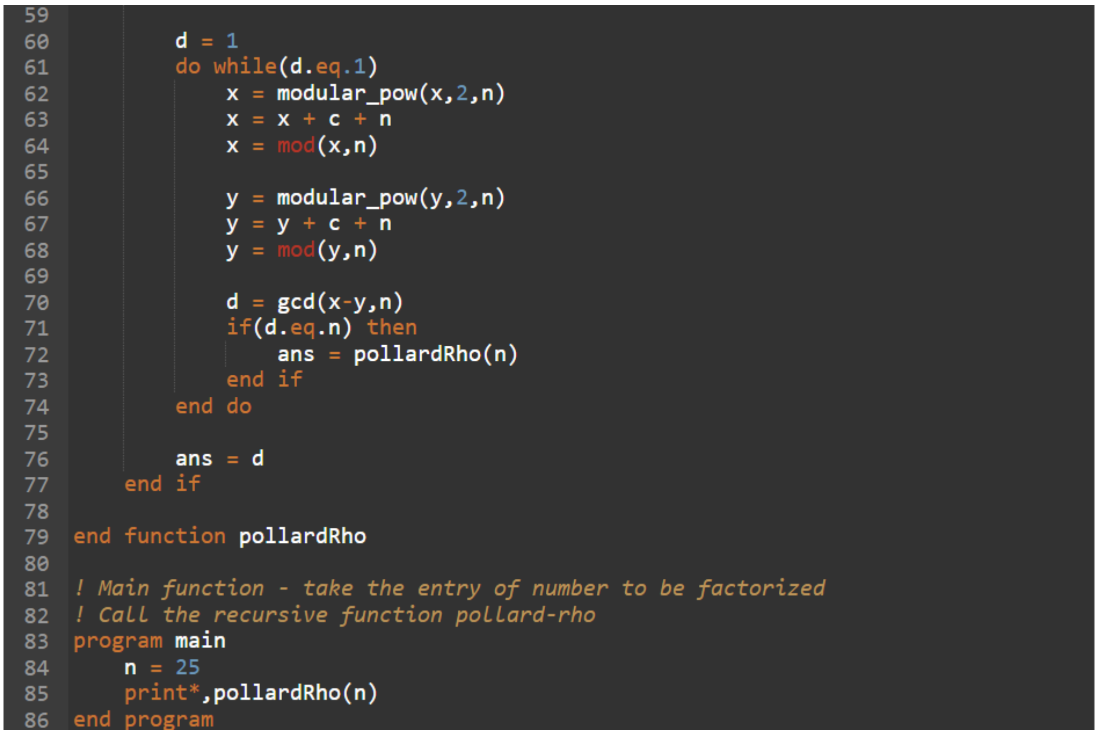
```fortran
! This function is used for moudlar exponentiation and returns the result as res
function modular_pow(base,exponent,modulus) result(res)
    integer :: exponent,base,modulus
    integer :: res
    res = 1
    do while(exponent.gt.0)
        if(mod(exponent,2).eq.1) then
            res = res*base
            res = mod(res,modulus)
        end if
        ! here rshift will right shift the exponent by 1
        exponent = rshift(exponent,1)
        base=base*base
        base=mod(base,modulus)
    end do

end function modular_pow
```

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



```fortran
! This is the recursive function
! This is the main function for pollard-rho factorization
recursive function pollardRho(n) result(ans)
    integer :: x,y,c,d
    real :: r

    if(n.eq.1) then
        ans=n
    end if

    if(mod(n,2).eq.0) then
        ans=2
    end if

    if(ans.ne.n .and. ans.ne.2) then

        ! random_number will generate a random number between 0 and 1
        ! We have applied the upper and lower limits to limit the random number
        ! between a and b
        call random_number(r)
        x = (2-0+1)*r + 0
        x = mod(x,n-2)

        y = x
        call random_number(r)
        c = (1-0+1)*r + 0
        c = mod(c,n-1)
```

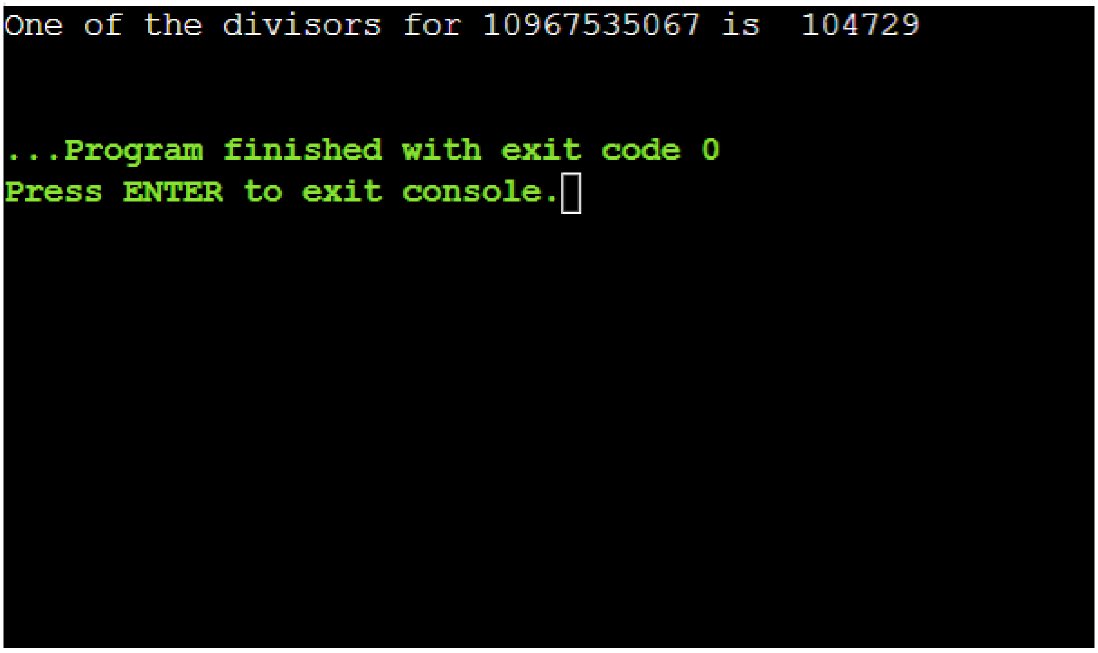
FIGURE 2.7: Part 3 of FORTRAN Code of Pollard Rho Factorization



```fortran

        d = 1
        do while(d.eq.1)
            x = modular_pow(x,2,n)
            x = x + c + n
            x = mod(x,n)

            y = modular_pow(y,2,n)
            y = y + c + n
            y = mod(y,n)

            d = gcd(x-y,n)
            if(d.eq.n) then
                ans = pollardRho(n)
            end if
        end do

        ans = d
    end if

end function pollardRho

! Main function - take the entry of number to be factorized
! Call the recursive function pollard-rho
program main
    n = 25
    print*,pollardRho(n)
end program
```

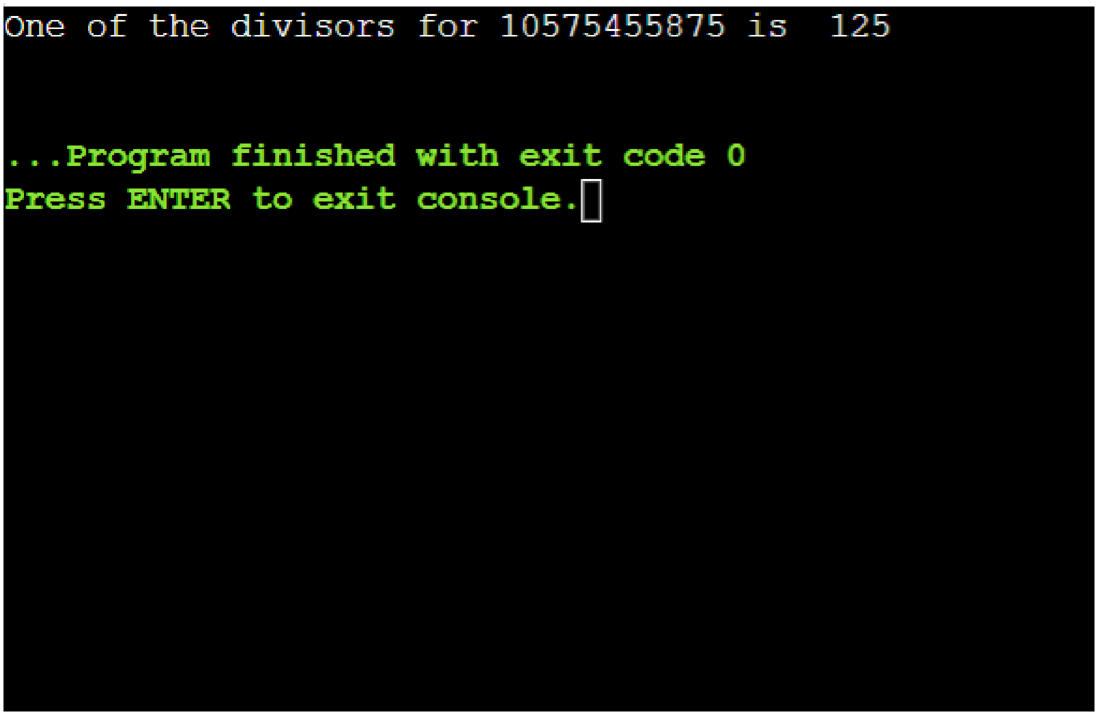
FIGURE 2.8: Part 4 of FORTRAN Code of Pollard Rho Factorization

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

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 :

Proposition 1 : Let S be a set of r elements. Given a map f from S to S and and element $x_0 \in$ S, let x_{j+1} = f(x_j) for j=0,1,2... 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, ..., 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 $e^{-\lambda}$. [1]

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 $O((n)^{\frac{1}{4}}.log^2n)$.
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 λ = 9. Then we know that for an average pair (f,x_0) we are almost certain to factor n in $3C(n)^{\frac{1}{4}}log^2n$ bit operations. The **space complexity** is O(N) due to recursive stack space. [1]

2.3 Pollard p-1 Factorization

2.3.1 Introduction

Pollard p-1 is a prime factorization algorithm for integers. Pollard p-1 algorithm is invented by John Pollard in 1974. Pollard p-1 algorithm is a special purpose algorithm which can be used for integers 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 cryptographic protocols. [15] The concepts used in Pollard p-1 factorization algorithm is : Len 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 :

$$a^{K(p-1)} \equiv 1(modp) \tag{2.3}$$

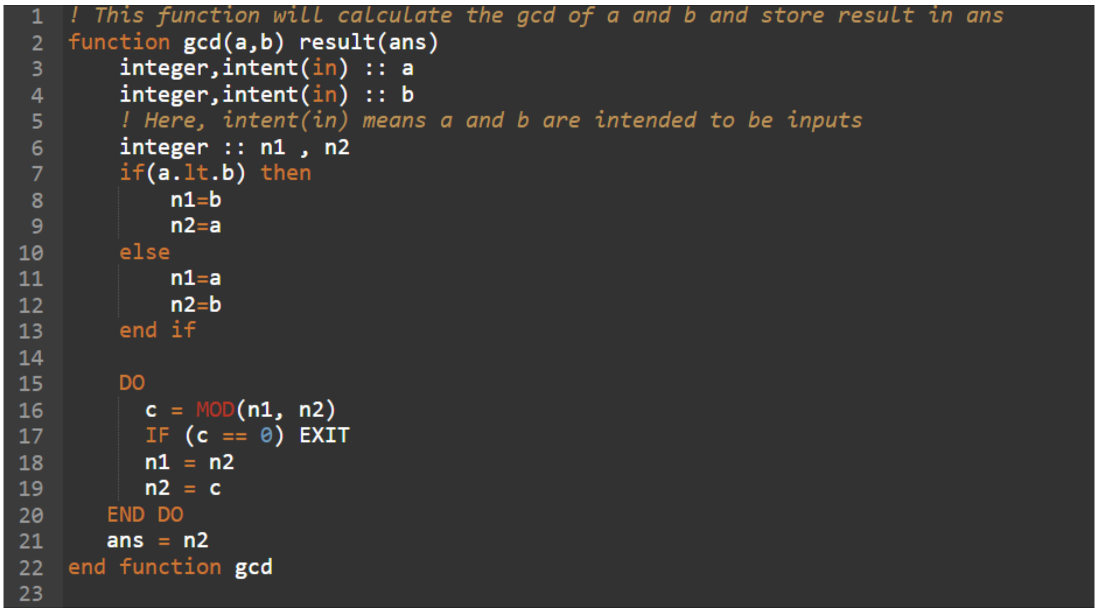
Pollard p-1 method is a classical factoring technique. Suppose we want to factor the composite number n, and p is some prime factor of n. If p has the property that p-1 has no large prime divisor, then this pollard p-1 method is virtually certain to find p. [1]

2.3.2 Algorithm

Algorithm 11 Pollard p-1 Factorization [1]

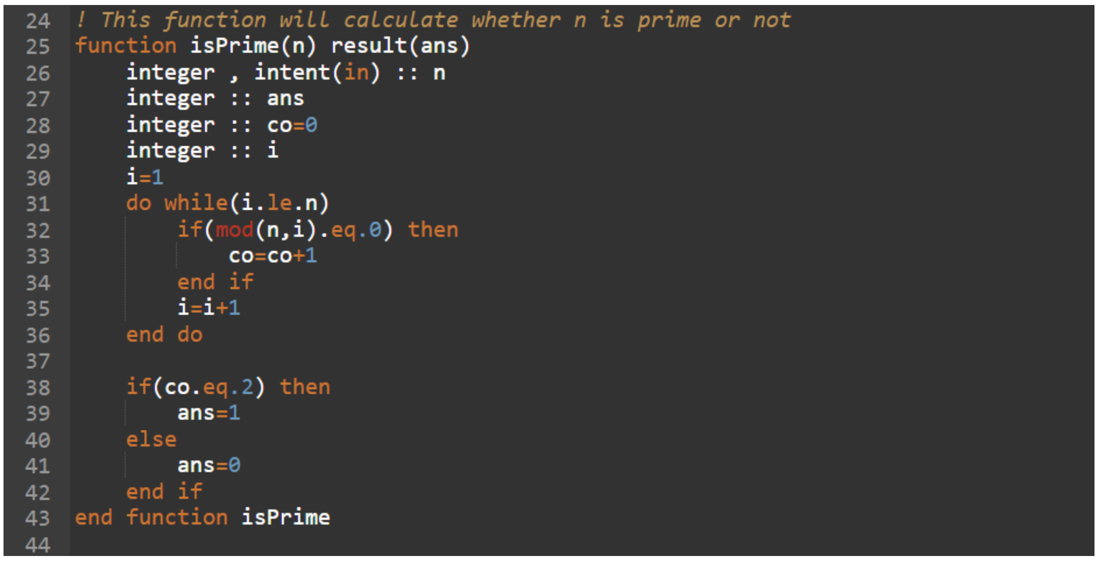
```
1: procedure POLLARD-P-1($N$)        ▷ Input is n∈N which needs to be factorized
2:     System Initialization
3:     Read the value of n
4:     Choose an integer k which is multiple of all or most integers less than some bound B.
5:     Choose an integer a between 2 and n-2. a could be any randomly chosen integer.
6:     Compute $a^k$ mod n by repeated squaring method.
7:     Compute 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



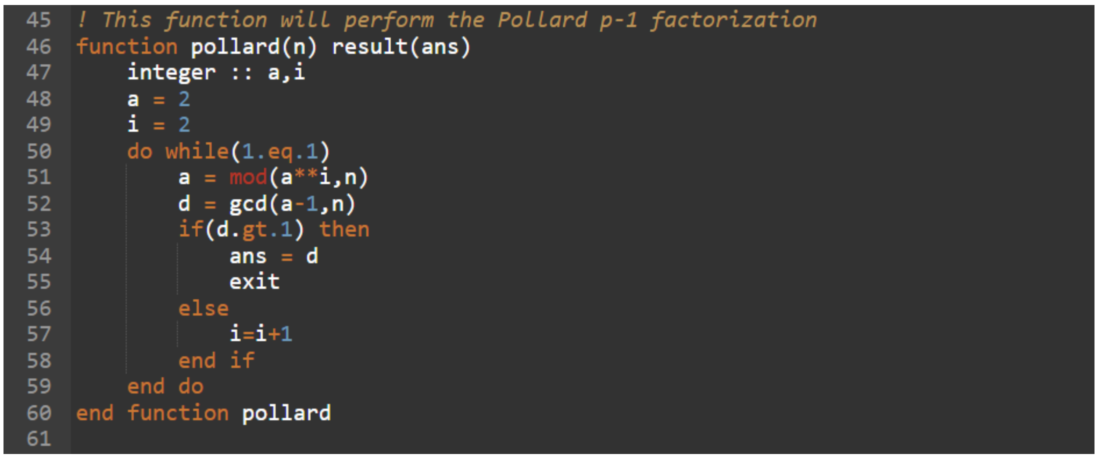
```fortran
! This function will calculate the gcd of a and b and store result in ans
function gcd(a,b) result(ans)
    integer,intent(in) :: a
    integer,intent(in) :: b
    ! Here, intent(in) means a and b are intended to be inputs
    integer :: n1 , n2
    if(a.lt.b) then
        n1=b
        n2=a
    else
        n1=a
        n2=b
    end if

    DO
      c = MOD(n1, n2)
      IF (c == 0) EXIT
      n1 = n2
      n2 = c
   END DO
   ans = n2
end function gcd

```

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



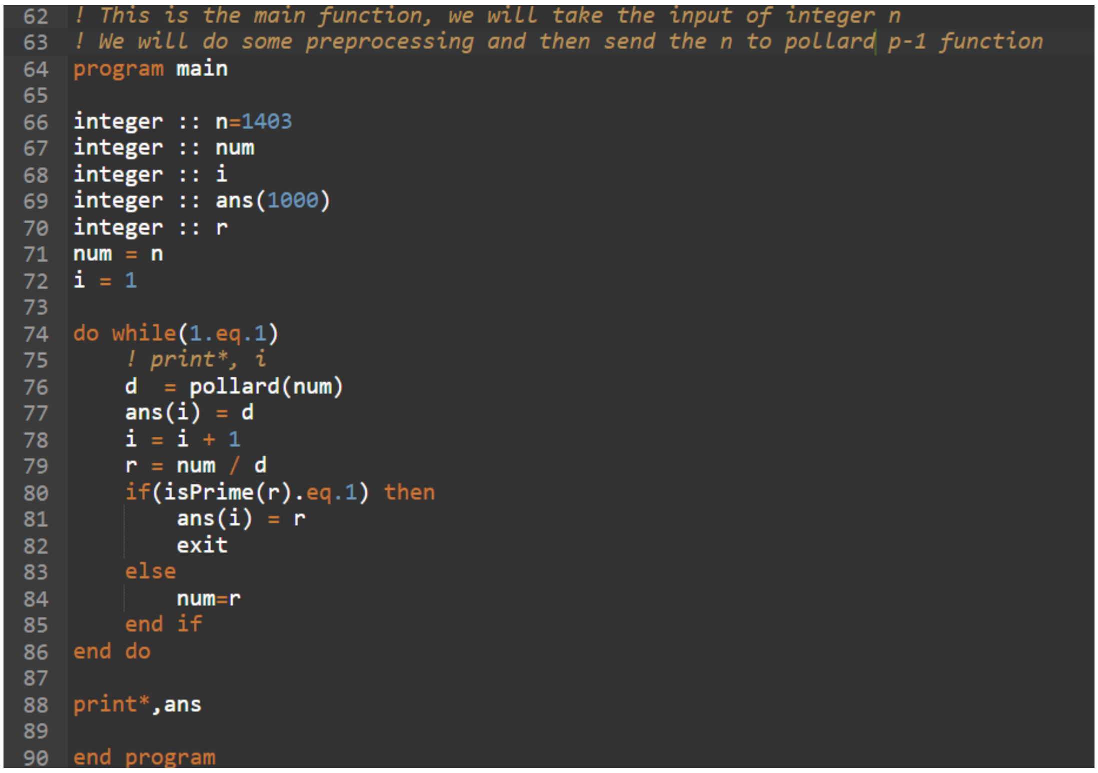
```fortran
! This function will calculate whether n is prime or not
function isPrime(n) result(ans)
    integer , intent(in) :: n
    integer :: ans
    integer :: co=0
    integer :: i
    i=1
    do while(i.le.n)
        if(mod(n,i).eq.0) then
            co=co+1
        end if
        i=i+1
    end do

    if(co.eq.2) then
        ans=1
    else
        ans=0
    end if
end function isPrime

```

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



```fortran
! This function will perform the Pollard p-1 factorization
function pollard(n) result(ans)
    integer :: a,i
    a = 2
    i = 2
    do while(1.eq.1)
        a = mod(a**i,n)
        d = gcd(a-1,n)
        if(d.gt.1) then
            ans = d
            exit
        else
            i=i+1
        end if
    end do
end function pollard

```

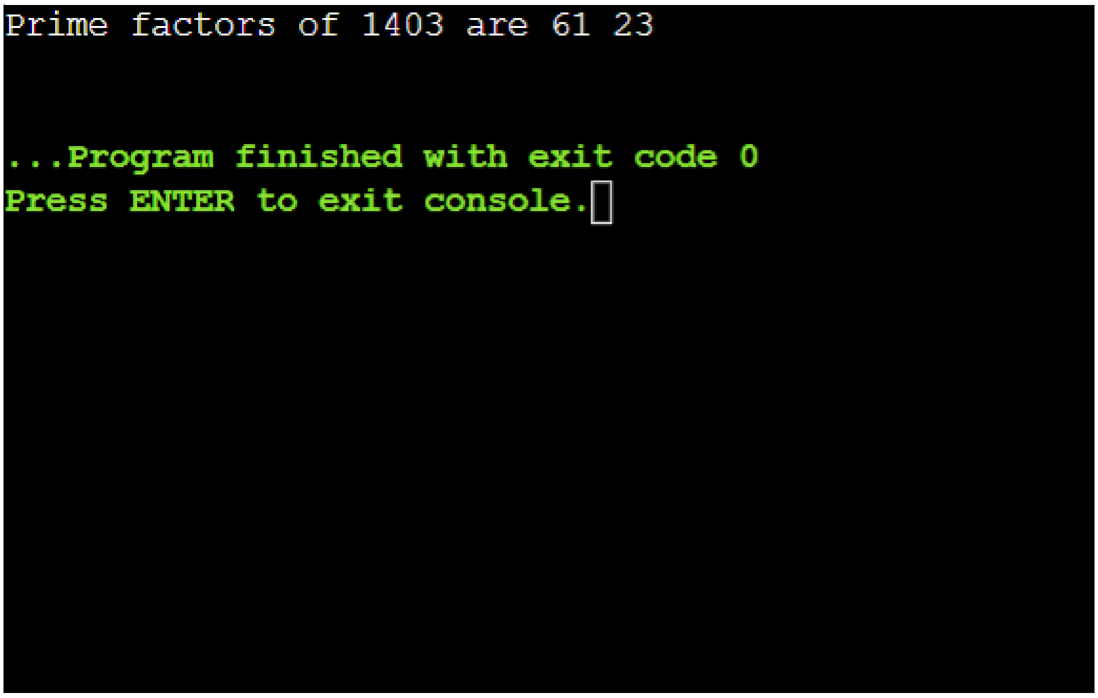
FIGURE 2.13: Part 3 of FORTRAN Code of Pollard p-1 Factorization



```fortran
! This is the main function, we will take the input of integer n
! We will do some preprocessing and then send the n to pollard p-1 function
program main

integer :: n=1403
integer :: num
integer :: i
integer :: ans(1000)
integer :: r
num = n
i = 1

do while(1.eq.1)
    ! print*, i
    d  = pollard(num)
    ans(i) = d
    i = i + 1
    r = num / d
    if(isPrime(r).eq.1) then
        ans(i) = r
        exit
    else
        num=r
    end if
end do

print*,ans

end program
```

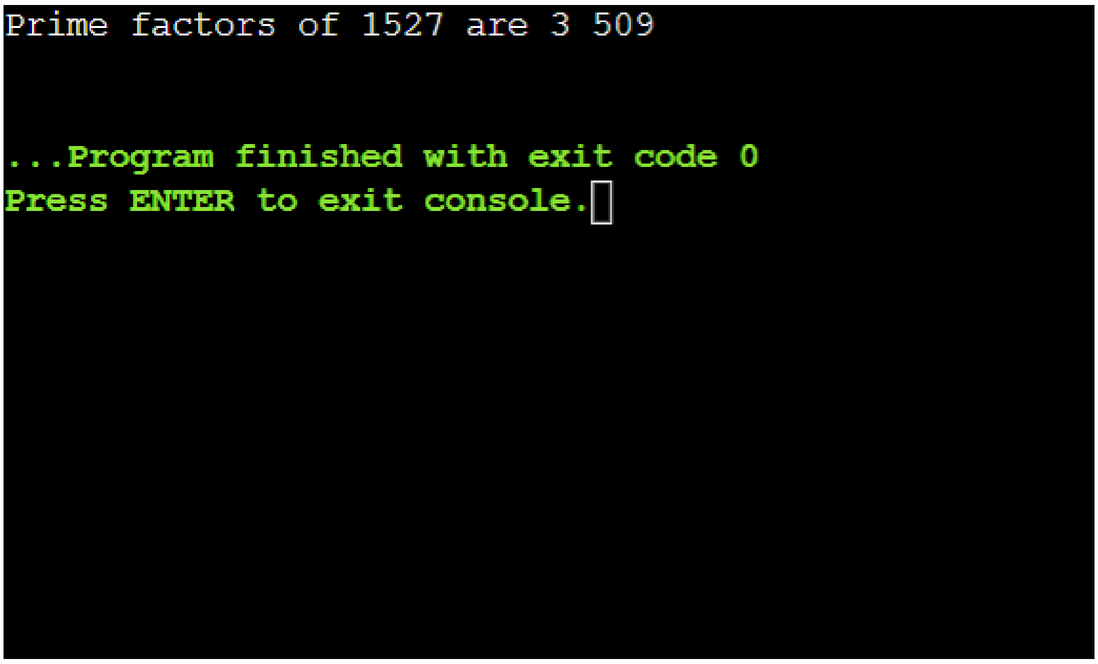
FIGURE 2.14: Part 4 of FORTRAN Code of Pollard p-1 Factorization

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 $O(n^2log^3(n))$.
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 $O(B^2log^2(B))$, or we can say it is bounded by $O(n^2log^2n)$.
2. In step 3, we have described the method to compute a^k mod 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 $O(n^2log^3(n))$ and the **space complexity** of pollard p-1 algorithm would be O(1) since no extra space is used in the code. [1]

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 cryptography. 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 Helman key 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] Anuj Jakhar, “Primality and factoring,” pp. 1–142, Indian Institute of Science Education and Research, April 25, 2014.

[2] http://www.cs.cornell.edu/, 2023.

[3] KaashyapMSK, “How is the time complexity of Sieve of Eratosthenes is n*log(log(n))?.” https://www.geeksforgeeks.org/how-is-the-time-complexity-of-sieve-of-eratosthenes-is-nloglogn/, 24 Jan, 2023.

[4] https://www.geeksforgeeks.org/aks-primality-test/, 01 Sep, 2022.

[5] https://archive.org/, 01 Sep, 2022.

[6] Surya Priy, “AKS Primality Test.” https://www.geeksforgeeks.org/aks-primality-test/, 01 Sep, 2022.

[7] Roeland Singer-Heinze, “Run time efficiency and the aks primality test,” *Thesis*, pp. 1–18, May 30, 2013.

[8] https://docplayer.net/, May 30, 2013.

[9] “Lucas–Lehmer primality test.” https://en.wikipedia.org/wiki/LucasLehmer_primality_test.

[10] http://www.seil-oi.eu/.

[11] “Fermat primality test.” https://en.wikipedia.org/wiki/Fermat_primality_test.

[12] https://www.coursehero.com/.

[13] GeeksforGeeks, “Lucas Primality Test.” https://www.geeksforgeeks.org/lucas-primality-test/, 29 Jul, 2022.

[14] “Pollard’s rho algorithm.” https://en.wikipedia.org/wiki/Pollard's_rho_algorithm.

[15] “Pollard’s p - 1 algorithm.” https://en.wikipedia.org/wiki/Pollard'sp-1algorithm.