“ALGORITHM FOR INTEGER FACTORIZATION”

PROJECT REPORT

Submitted for CAL in B.Tech Data Structure and Algorithm (CSE2003)

By

SHIVAM KAPOOR (15BCE1339)
V. MADHU SAMHITHA (15BCE1193)
TANMAYA SANGWAN (15BCE1177)

Slot: L9 – L19

Name of faculty: **Prof. Nagaraj S.V.**

(SCHOOL OF COMPUTING SCIENCE AND ENGINEERING)

MAY, 2016

CERTIFICATE

This is to certify that the Project work entitled ***"ALGORITHM FOR INTEGER FACTORIATION"*** that is being submitted by ***" SHIVAM KAPOOR (15BCE1339), V. MADHU SAMHITHA (15BCE1193) and TANMAYA SANGWAN (15BCE1177) "*** for CAL in B.Tech Data Structure and Algorithms (CSE2003) is a record of bonafide work done under my supervision. The contents of this project work have not been submitted for any other CAL course.

Place : Chennai

Date : 01/05/2016

SIGNATURE OF STUDENTS: SHIVAM KAPOOR (15BCE1339)

V. MADHU SAMHITHA (15BCE1193)

TANMAYA SANGWAN (15BCE1177)

SIGNATURE OF FACULTY: PROF. NAGARAJ S. V.

____

ACKNOWLEDGEMENT

First of all, we would like to express our gratitude to the SCSE Dean & the University Management for giving us an opportunity to carry out this project and prove our worth.

We would also like to thank our families and friends for moral support and constant guidance.

And finally, this project would not have been possible without the constant guidance and support offered by our Professor Dr. Nagaraj S.V. His suggestions and advices helped us a lot.

SHIVAM KAPOOR (15BCE1339)
V. MADHU SAMHITHA (15BCE1193)
TANMAYA SANGWAN (15BCE1177)

ABSTRACT

The problem of integer factorization has been around for a very long time. This report describes a number of algorithms and methods for performing factorization. The problem of factorization has been known for thousands of years. However, only recently did it become "popular". This sudden interest in factorization was due to the advances in cryptography, and mainly the RSA public key cryptosystem. There are a lot of factorization algorithms out there. Some of them are heavily used, others just serve educational purposes.

This project is aimed to research on the above. In this report, we have studied, analyzed and implemented many of the common factorization algorithms out there. Along with the algorithm following topics are covered –

- Time Complexity.
- Data Structures Used.
- Proof of correctness.
- Code Implementation.

Also, references are attached at the end of the document.

CONTENTS

INTRODUCTION

In number theory, ***integer factorization*** is the decomposition of a composite number into a product of smaller integers. If these integers are further restricted to prime numbers, the process is called prime factorization.

When the numbers are very large, no efficient, non-quantum integer factorization algorithm is known. An effort by several researchers, concluded in 2009, to factor a 232-digit number (RSA-768) utilizing hundreds of machines took two years and the researchers estimated that a 1024-bit RSA modulus would take about a thousand times as long. However, it has not been proven that no efficient algorithm exists. Many areas of mathematics and computer science have been brought to bear on the problem, including elliptic curves, algebraic number theory, and quantum computing.

The problem of factorization has been known for thousands of years. However, only recently did it become "popular". This sudden interest in factorization was due to the advances in cryptography, and mainly the RSA public key cryptosystem. There are a lot of factorization algorithms out there. Some of them are heavily used, others just serve educational purposes. The factorization algorithms may be distinguished in two different ways:

- Deterministic or nondeterministic
- Run time depends on size of N or f.

Deterministic algorithms are algorithms which are guaranteed to find a solution if we let them run long enough. On the contrary, nondeterministic algorithms may never terminate. The most usual distinction, however, deals with the runtime of the algorithm. **The running time of recent algorithms depends on the size of the input number N, whereas older algorithms depended on the size of the factor f which they find.**

OUTLINE OF WORK

- Work was equally divided among the group members. Equal number of algorithms were studied, analyzed and implemented by each individual.
- In accordance to the above, proof of correctness and time complexity were also computed for each and every algorithm.
- Algorithms were designed so to have the least time complexity possible.
- Implementation of codes are done in several coding languages so as to show the versatility of the algorithm.

SOFTWARES USED

- **Python IDLE -** version : 3.4.3 , Tk version: 8.6.1
- **Code::Blocks** - version: Release 13.12 rev 9501

  SDK Version – 1.19.0
- **MATLAB R2014a –** version: 8.3.0.532 64-bit
- **Code Chef Java** – Javac 8

Part - I

DOCUMENTATION

ALGORITHM 1
TRIAL AND DIVISION METHOD

1.1 Introduction

This algorithm essentially tries to factorize an integer N using “brute force”. Starting at p = 2, this algorithm tries to divide N with every number until it succeeds. When this happens, it sets N←N=p, and resumes its operation. The way in which we choose our p can speed up, or slow down, our algorithm. For instance, we could pick our p’s sequentially, by adding 1 at every iteration.

1.2 Algorithm

```
INPUT N
test factor = 2
WHILE (N > 1 AND test factor < max)
    IF (N MOD test factor) == 0 THEN
        N = N / test factor
        PRINT test factor
    ELSE
        test factor := test factor + 1
```

1.3 Data Structure Used

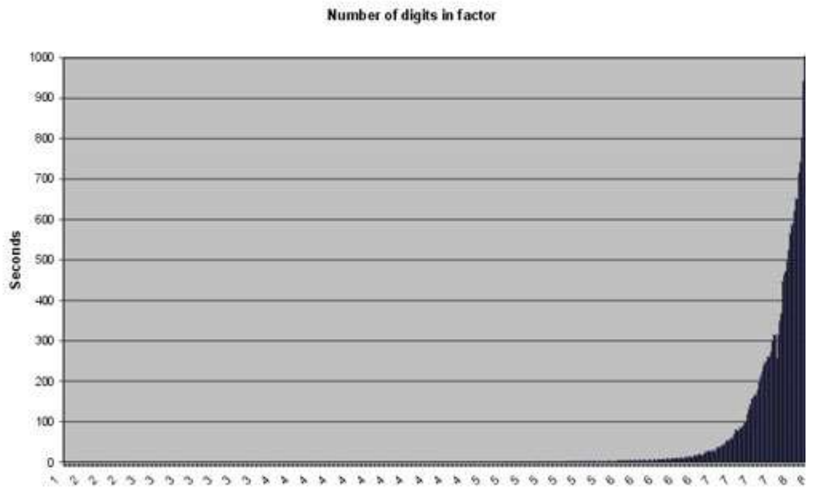
No specific data structure used. Normal syntaxes and data types are used.

1.4 Time Complexity

The expected running time of this algorithm is:

O (f . (log N)2)

Where f is the size of the factor found.



As expected, the amount of time the algorithm takes increases exponentially with the size of the factor found. Practically, after 6 or 7 digits, this algorithm becomes too expensive.

1.5 Proof of correctness

This algorithm can be proved for correctness by the theorem of induction. Basically we assume that the algorithm will work for an input k, thus the algorithm will be correct if it will also work for input k+1.

1.6 Code and output

Code for this algorithm is implemented in Python 3.4.3
For code, see Appendix.

ALGORITHM 2
TRIAL AND DIVISION METHOD (Efficient)

2.1 Introduction

It is an improvement of the above algorithm. If we look carefully, all the divisors are present in pairs. For example if n = 100, then the various pairs of divisors are: (1,100), (2, 50), (4, 25), (5, 20), (10, 10)

Using this fact we could speed up our program significantly.
We, however have to careful if there are two equal divisors as in case of (10, 10). In such case we'd print only one of them.

2.2 Algorithm

```
ALGORITHM (pseudocode):
Function trialDivision (N)
For i from 2 to floor (sqrt (N))
   If s divides N then
       Return s, N/s
   End if
End for
```

2.3 Data Structure Used

No specific data structure used. Normal syntaxes and data types are used.

2.4 Time Complexity

The expected running time of this algorithm is:

O (sqrt (n))

This algorithm is comparatively very efficient than the normal trial and division algorithms. This algorithm can be used to remove small factors of a big number so that the umber can be further factorized by other general, more efficient algorithms.

2.5 Proof of correctness

This algorithm can be proved for correctness by the theorem of induction. Basically we assume that the algorithm will work for an input k, thus the algorithm will be correct if it will also work for input k+1.

2.6 Code and output

Code for this algorithm is implemented in Python 3.4.3 and Matlab R2014a. For code, see Appendix.

ALGORITHM 3
POLLARD’S RHO METHOD

3.1 Introduction

Pollard's rho algorithm is a special-purpose integer factorization algorithm. It was invented by John Pollard in 1975. It is particularly effective for a composite number having a small prime factor.

Pollard's Rho Algorithm is a very interesting and quite accessible algorithm for factoring numbers. It is not the fastest algorithm by far but in practice it outperforms trial division by many orders of magnitude. It is based on very simple ideas that can be used in other contexts as well.

3.2 Algorithm

The algorithm takes as its inputs n, the integer to be factored; and $g(x)$, a polynomial $p(x)$ computed modulo n. This will ensure that if $p|n$, and $x \equiv y \bmod p$, then $g(x) \equiv g(y) \bmod p$. In the original algorithm, $g(x) = (x^2 - 1) \bmod n$, but nowadays it is more common to use $g(x) = (x^2 + 1) \bmod n$.

In short, the algorithm comprises of the following steps:
1. Construct a sequence of integers which is periodically recurrent (mod p), where p is a prime factor of N.
2. Search for the period of repetition, i.e. find i and j such that xi is congruent to xj (mod p).
3. Calculate the factor p of N.

The output is either a non-trivial factor of n, or failure. It performs the following steps:

```
x ← 2; y ← 2; d ← 1
while d = 1:
    x ← g(x)
    y ← g(g(y))
    d ← gcd(|x - y|, n)
if d = n:
    return failure
else:
    return d
```

3.3 Data Structure Used

No specific data structure used. Normal syntaxes and data types are used.

3.4 Time Complexity

If the pseudo random number x = g(x) occurring in the Pollard ρ algorithm were an actual random number, it would follow that success would be achieved half the time, by the Birthday paradox in $O(\sqrt{p}) \leq O(n^{1/4})$ iterations. It is believed that the same analysis applies as well to the actual rho algorithm, but this is a heuristic claim, and rigorous analysis of the algorithm remains open.

3.5 Proof of correctness

Pollard's Rho Method is not guaranteed to work. In particular the chance of Pollard's Rho Method failing after i iterations is e^((1 – i^2)/2r) where r is some factor of n less than sqrt(n), and i is the number of iterations used.

The Pollard Rho algorithm is not a fast one, running in expected time proportional to the square root of the factor found, which is usually (but not always) the smallest factor. However, this effectively relies on the birthday problem, and is only a heuristic, not proven. It's also not guaranteed to factor the number at all.

3.6 Code and output

Code for this algorithm is implemented in C. For code, see Appendix.

ALGORITHM 4
FERMAT's FACTORIZATION METHOD

4.1 Introduction

It Fermat's factorization method, named after Pierre de Fermat, is based on the representation of an odd integer as the difference of two squares:

$$N = a^2 - b^2.$$

That difference is algebraically factorable as $(a+b)(a-b)$; if neither factor equals one, it is a proper factorization of N. Each odd number has such a representation. Indeed, if $N = cd$ is a factorization of N, then

$$N = \left(\frac{c+d}{2}\right)^2 - \left(\frac{c-d}{2}\right)^2$$

Since N is odd, then c and d are also odd, so those halves are integers. (A multiple of four is also a difference of squares: let c and d be even.)In its simplest form, Fermat's method might be even slower than trial division (worst case). Nonetheless, the combination of trial division and Fermat's is more effective than either.

4.2 Algorithm

```
INPUT N
sqrt: = [√ N]
u := 2 * sqrt + 1
v := 1
r := sqrt * sqrt - N
WHILE r <> 0
    IF r > 0 THEN /* Keep increasing y */
        WHILE r > 0
              r := r - v
              v := v + 2
    IF r < 0 THEN /* Increase x */
         r := r + u
         u := u + 2
PRINT (u + v - 2) / 2
PRINT (u - v) / 2
```

4.3 Data Structure Used

No specific data structure used. Normal syntaxes and data types are used.

4.4 Time Complexity

How much work is actually needed to find the factors of N? Let us suppose that N = a × b, with a < b. The factorization will be achieved when x = (a + b)/2. Since the starting value of x is √ N, and b = N/a, the factorization will take approximately

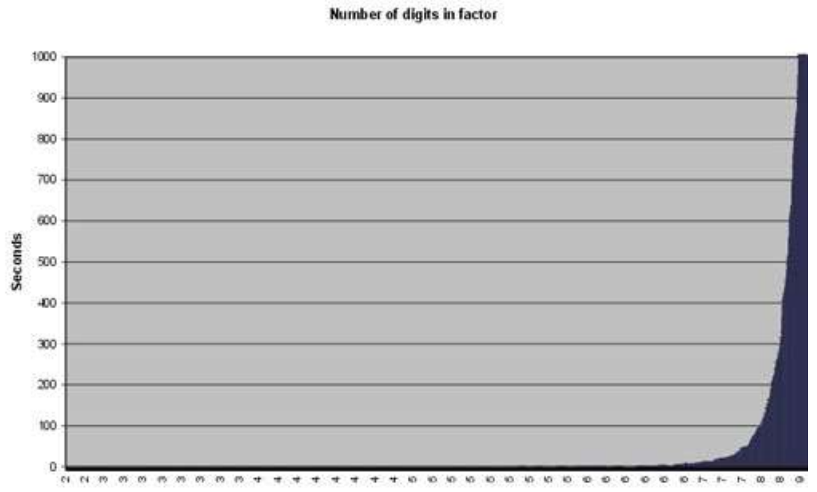
$$1/2\ (a + N/a\ ) - \sqrt{N} = (\sqrt{N} - a)^2 / 2a$$

cycles. If the two factors of N are really close, i.e. if a = k √ N, with 0<k<1, then the number of cycles required in order to obtain the factorization is (1 − k)2 /2k . √ N.

This complexity is of the order O(cN1/2). However, the value of k can be very small, and thus making this algorithm impractical. For instance, let us consider an “ordinary” case where a ≈N1/3 and b ≈ N2/3. In such a case, the number of cycles necessary will be

$$\frac{(\sqrt{N} - \sqrt[3]{N})^2}{2\sqrt[3]{N}} = \frac{(\sqrt[3]{N})^2(\sqrt[6]{N} - 1)^2}{2\sqrt[3]{N}} \approx \frac{1}{2}N^{\frac{2}{3}},$$

Which is considerably higher than O (N1/2). Therefore this algorithm is only practical when the factors a and b are almost equal to each other.



Additionally, this algorithm suffers from the same problem as trial divisions, it will prove primality in the worst case. If this algorithm is given a prime number p, then the results will eventually be 1 and p. By the way, this is even worse than proving primality with trial divisions. The total number of cycles required for proving primality is $n - \sqrt{n}$, which is much worse than trial divisions.

4.5 Efficiency of Fermat's Algorithm

Notice that Fermat's method is very fast if $n = pq$, where $p \approx q \approx \sqrt{n}$. Because of this fact, RSA primes are chosen carefully to not have this property, just as they are chosen to not have very small factors (which are easy to find with trial division). For this reason, Fermat's method is not as efficient for breaking crypto schemes as it looks at first glance.

In fact, as the distance between p and $\sqrt{n}$ increases, the running time increases faster than exponentially. So, with just a tiny bit of foresight, it is easy to design an RSA modulus that makes Fermat's method computationally infeasible.

4.6 Code and output

Code for this algorithm is implemented in C. For code, see Appendix.

ALGORITHM 5
POLLARD’S p-1 METHOD

5.1 Introduction

Pollard's p − 1 algorithm is a number theoretic integer factorization algorithm, invented by John Pollard in 1974. It is a special-purpose algorithm, meaning that it is only suitable for integers with specific types of factors; it is the simplest example of an algebraic-group factorization algorithm.
This algorithm is pretty much based on Fermat’s little theorem: If p is prime, and a ≠ 0 mod p then -

$$a^{p-1} \equiv 1 \pmod{p}:$$

5.2 Algorithm

```
INPUT N, c, max
m := c
FOR i := 1 to max DO
    m := modexpo(m, i, N)
      IF (i MOD 10 == 0) THEN
              g := gcd(m-1, N)
              IF g > 1 THEN
                  PRINT g
```

5.3 Data Structure Used

No specific data structure used. Normal syntaxes and data types are used.

5.4 Time Complexity

In the worst case, Pollard’s p - 1 algorithm takes as long as -

O (f . (log N)2)

Where f is the size of factor found.

However, it usually does better, provided that we are lucky enough to find a factor.
Apparently, Pollard's p - 1 algorithm is much faster that Pollard's p algorithm, but with less success. It turns out that the algorithm seldomly gives back results, but when it does, it is very fast

5.5 Proof of correctness

Pollard's p – 1 has the same problems as the pollard's rho. As described earlier, at some point we might find the GCD to be equal to N. In such cases we will want to try to change the base a to a different integer. Also, the algorithm might not terminate if p - 1 has only large prime factors.

It has been statistically found that the largest prime factor of an arbitrary integer N usually falls around N0.63. Therefore, with a limit of 10000, Pollard p - 1 will find prime factors that are less than two million. We should keep in mind however that there is a fairly wide distribution of the largest prime factor of N, and therefore factors much larger than two million may be found.

The largest factor found by this algorithm during the Cun-ningham project is a 32-digit factor

49858990580788843054012690078841

of 2977 - 1.
Also, Because of Pollard p-1, the RSA public key crypto-system has restrictions on the primes a and b that are chosen. Essentially, if a - 1 or b - 1 have only small prime factors, then Pollard p - 1 will break the encryption very quickly.

5.6 Code and output

Code for this algorithm is implemented in C++ . For code, see Appendix.

ALGORITHM 6
EULER'S METHOD OF FACTORIZATION

6.1 Introduction

Euler's factorization method is a technique for factoring a number by writing it as a sum of two squares in two different ways.
The idea that two distinct representations of an odd positive integer may lead to a factorization was apparently first proposed by Marin Mersenne. However, it was not put to use extensively until Euler one hundred years later. His most celebrated use of the method that now bears his name was to factor the number 1000009 , which apparently was previously thought to be prime even though it is not a pseudo prime by any major primality test.

Euler's factorization method is more effective than Fermat's for integers whose factors are not close together and potentially much more efficient than trial division if one can find representations of numbers as sums of two squares reasonably easily.

The great disadvantage of Euler's factorization method is that it cannot be applied to factoring an integer with any prime factor of the form 4k + 3 occurring to an odd power in its prime factorization, as such a number can never be the sum of two squares. Even odd composite numbers of the form 4k + 1 are often the product of two primes of the form 4k + 3 (e.g. 3053 = 43 × 71) and again cannot be factored by Euler's method.

6.2 Algorithm

```
count := 0
i := 0
INPUT num;
x := round(√num/2)
y :=x
WHILE x>0 && y>0 && x<=sqrt(num) && y<=sqrt(num)
    WHILE y<=sqrt(num)
```

```
        WHILE x>0
            sqr := x*x + y*y
            IF (sqr==num) THEN
                    array[i] := x
                    array[i+1] := y
                    i := i+2
                    count++
             x--
        y++;
        x := round(sqrt(num/2));

temp1 := abs(array[0] - array[2])
temp2 := abs(array[1] - array[3])
k := gcd(temp1,temp2)
l := temp1/k
m := temp2/k
n := (array[0] + array[2])/m

factor1 := (0.5*k)^2 + (0.5*n)^2
factor2 := l^2 + m^2
PRINT factor1 and factor2.
```

6.3 Data Structure Used

No specific data structure used. Normal syntaxes and data types are used.

6.4 Time Complexity

The time complexity for this program can be easily calculated as there are 3 while loops and for 2 loops, time complexity is sqrt(n) and for one it is n. Therefore,

T(n)= O(sqrt(n)*sqrt(n)*sqrt(n/2))

T(n) = O((n^1.5)/sqrt(2))

Hence the algorithm is of time complexity **O((n^1.5)/sqrt(2))**

6.5 Proof of correctness

Given n is the number we have to factorise, then, we can express n as two sums of two squares, i.e if n = a^2 + b^2 = c^2 + d^2, Euler states that we can find n as a product of sums of two squares.

First deduce that $a^2 - c^2 = d^2 - b^2$
and factor both sides to get

$$(a-c)(a+c) = (d-b)(d+b) \quad ....(1)$$

Now let $k = \gcd(a-c,d-b)$ and $h = \gcd(a+c,d+b)$ so that there exists some constants l,m,l',m' satisfying

$$(a-c) = kl,$$
$$(d-b) = km,$$

where $\gcd(l,m) = 1$

$$(a+c) = hm',$$
$$(d+b) = hl',$$

where $\gcd(l',m') = 1$

Substituting these into equation (1) gives

$$klhm' = kmhl'$$

Canceling common factors yields

$$lm' = l'm$$

Now using the fact that (l,m) and (l',m') are pairs of relatively prime numbers, we find that

$$l = l'$$
$$m = m'$$

So

$$(a-c) = kl$$
$$(d-b) = km$$
$$(a+c) = hm$$
$$(d+b) = hl$$

We now see that $m = \gcd(a+c,d-b)$ and $l = \gcd(a-c,d+b)$

Applying the Brahmagupta–Fibonacci identity (i.e, that the product of two sums of two squares is a sum of two squares) ,we get

$$(k^2 + h^2)(l^2 + m^2) = (kl - hm)^2 + (km + hl)^2 = ((a-c) - (a+c))^2 + ((d-b) + (d+b))^2 = (2c)^2 + (2d)^2 = 4n,$$

$$(k^2 + h^2)(l^2 + m^2) = (kl + hm)^2 + (km - hl)^2 = ((a-c) + (a+c))^2 + ((d-b) - (d+b))^2 = (2a)^2 + (2b)^2 = 4n.$$

As each factor is a sum of two squares, one of these must contain both even numbers: either (k, h) or (l ,m). Without loss of generality, assume that pair (k,h) is even. The factorization then becomes

$$n = ((k/2)^2 + (h/2)^2)*(l^2 + m^2))$$

Hence we prove the algorithm

6.6 Code and output

Code for this algorithm is implemented in C++ .For code, see Appendix.

ALGORITHM 7
DIXON’S FACTORIZATION METHOD

7.1 Introduction

In number theory, **Dixon's factorization method** (also Dixon's random squares method or Dixon's algorithm) is a general-purpose integer factorization algorithm; it is the prototypical factor base method. Unlike for other factor base methods, its run-time bound comes with a rigorous proof that does not rely on conjectures about the smoothness properties of the values taken by polynomial.

Dixon's method is based on finding a congruence of squares modulo the integer N which we intend to factor. Fermat's factorization algorithm finds such a congruence by selecting random or pseudo-random x values and hoping that the integer x2 mod N is a perfect square (in the integers):

$$x^2 \equiv y^2 \pmod{N}, \qquad x \not\equiv \pm y \pmod{N}.$$

For example, if N = 84923, we notice (by starting at 292, the first number greater than √N and counting up) that 5052 mod 84923 is 256, the square of 16. So (505 – 16)(505 + 16) = 0 mod 84923. Computing the greatest common divisor of 505 – 16 and N using Euclid's algorithm gives us 163, which is a factor of N.

7.2 Algorithm

```
CLASS Exe
    MAIN FUNCTION (String args[])
        READ N
        CALL dixon(n)
    END MAIN
```

```
GCD FUNCTION(double a,double b)
    IF(b==0) THEN
       RETURN a
    ELSEIF (b>a)
       RETURN gcd(b,a)
    ELSE
       RETURN gcd(b,a%b)

Checkprime FUNCTION(double n)
F := 0
  For  i=2 to i<=n/2
    IF(n%i==0)THEN
        F := 1
        if(f==1)
            RETURN 0;
        ELSE
            RETURN 1;
Dixon FUNCTION(double n)
    M := n^0.5
    C := floor(m)
    f1:= 0
    FOR LOOP f=c+1 to f<=n
        S := ((f*f)%n)^0.5
        m1 := floor(s)
        CHECK IF((s-m1)==0) THEN
            IF(s!=(f%n))
               P := f+s
               Q := f-s
               b1 := CALL checkprime(p)
            d1 := checkprime(q)
            CHECK IF(b1==1)THEN
               Z := gcd(q,n)
               b := checkprime(z)
            IF(b==1) THEN
               PRINT the factors.
            ELSE IF(d1==1) THEN
               z=gcd(p,n)
               b=checkprime(z)
            IF(b==1) THEN
                 PRINT the factors.
            ELSE
            f1 := 1
```

7.3 Data Structure Used

No specific data structure used. Normal syntaxes and data types are used.

7.4 Time Complexity

The optimal complexity of Dixon's method is

$$O\left(\exp\left(2\sqrt{2}\sqrt{\log n \log \log n}\right)\right)$$

7.5 Code and output

Code for this algorithm is implemented in Java. For code, see Appendix.

APPLICATIONS

1.1 Introduction

Integer factorization is a commonly used mathematical problem often used to secure public-key encryption systems. A common practice is to use very large semi-primes (that is, the result of the multiplication of two prime numbers) as the number securing the encryption. In order to break it, they would have to find the prime factorization of the large semi-prime number – that is, two or more **prime numbers** that multiplied together result in the original number.

1.2 Kid-RSA

Many cryptographic protocols are based on the difficulty of factoring large composite integers or a related problem—for example, the RSA problem. An algorithm that efficiently factors an arbitrary integer would render RSA-based public-key cryptography insecure.
To show how RSA works we have made a small basic code called Kid-RSA (K-RSA).

Code for this algorithm is implemented in MATLAB. For code, see Appendix.

Part - II

APPENDICES

CODES IMPLEMENTATION

- Trial and Division

  Python Implementation for this algorithm:-

```
"""
Python Program implementation to find the
factors of a number by trial and division
method.
"""

num = int(input("Enter the number to be
factorized: "))
print("The factors of",num,"are: ")
test_factor = 1
while( num > 1 and test_factor<= num ):
     if (num % test_factor == 0):
          print(test_factor)
     test_factor=test_factor+1
```

- Trial and Division (Efficient)

  1. Python Implementation for this algorithm:-

```
import math

def printDivisors(n):
    #Note that this loop runs till square root
    for i in range(1,int(math.sqrt(n))+2):
        if (n%i==0):
            #If divisors are equal, print only
one
            if (n/i == i):
                print("",i);
            else:       #Otherwise print both
                print("",i)
                print("",int(n/i))
```

```
"""Driver program to test above function"""

num = int(input("Enter the number to be
factorized: "))
print("The factors of",num,"are: ")
printDivisors(num);
```

2. MATLAB Implementation for this algorithm:-

```
clc
clear all
syms num i
num=input('Enter the number to be factorized:
');
fprintf('The factors of the given number are:
');
for i=1:(num^0.5)+2
    if mod(num,i)==0
        if num/i == i
            i
        else
            i
            num/i
        end
    end
end
```

- <u>Pollard's Rho Method</u>

  Implementation of this algorithm in C++ language:-

```
#include<iostream>
using namespace std;

int gcd(int a, int b)
{
    int remainder;
    while (b != 0)
    {
        remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
}
```

```
int main()
{
    int number, x_fixed = 2, cycle_size = 2, x
= 2, factor = 1;
    cout<<"Please enter the number to be
factorized: ";
    cin>>number;
    while (factor == 1)
    {
        for (int count=1;count <= cycle_size
&& factor <= 1;count++)
        {
            x = (x*x+1)%number;
            factor = gcd(x - x_fixed, number);
        }

        cycle_size *= 2;
        x_fixed = x;
    }
    cout << "\nThe factor is  " << factor;
}
```

- <u>Fermat's Method of Factorization</u>

  Implementation of this algorithm in C language:-

```
#include<stdio.h>
#include<math.h>
main()
{
    int num, sqrt, u, v, r, term1, term2,
term1a;
    printf("Please enter the number to be
factorized: ");
    scanf("%d",&num);
    sqrt = pow(num,0.5);
    u = (2*sqrt) + 1;
    v = 1;
    r = (sqrt * sqrt) - num;
    while(r != 0)
    {
        if (r>0)
            while (r>0)
            {
                r = r - v;
                v = v + 2;
            }
}
else
        {
            r = r + u;
            u = u + 2;
        }
    }
    term1 = (u + v - 2)/2;
    term2 = (u - v)/2;
    term1a = (term1 + term2)/2;
    printf("Value of a: %d\n", term1a);
    printf("Value of b: %d\n", term1 - term1a);
    printf("Factor 1: %d\n", term1);
    printf("Factor 2: %d", term2);
}
```

- <u>Pollard's p-1 Method</u>

  Implementation of this algorithm in C++ language:-

```
#include<iostream>
#include<math.h>
#define ll long long
using namespace std;

ll modexpo(ll base, ll exponent, int modulus)
{
    ll result = 1;
    while (exponent > 0)
    {
        if (exponent % 2 == 1)
            result = (result * base) % modulus;
        exponent = exponent >> 1;
        base = (base * base) % modulus;
    }
    return result;
}

ll gcd(ll a, ll b)
{
    ll remainder;
    while (b != 0)
    {
        remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
}
```

```
main()
{
    ll N,a,max,Q,result;
    int i;
    cout<<"Enter value of N:
    cin>>N;
    cout<<"Enter value of a: ";
    cin>>a;
    cout<<"Enter value of Max: ";
    cin>>max;
    Q = a;
    for(i=1;i<=max;i++)
    {
        Q=modexpo(Q,i,N);
        if (i%10 ==0)
        {
            result = gcd(Q-1, N);
            if (result>1)
            {
                cout<<result<<endl;
            }
        }
    }
}
```

- <u>Euler's Method of Factorization</u>

  Implementation of this algorithm is done in C++ -

```
#include<iostream>
#include<math.h>
#include<stdlib.h>
#define ll long long
using namespace std;

int gcd(int a, int b)
{
    int remainder;
    while (b != 0)
    {
        remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
}
```

```
main()
{
    ll num;
    int x,y,count=0,sqr,i=0;
    cout<<”Enter value of N: ”;
    x = round(sqrt(num/2));
    y=x;
    int array[10];
    while(x>0 && y>0 && x<=sqrt(num) &&
y<=sqrt(num))
    {
        while(y<=sqrt(num))
        {
            while(x>0)
            {
                sqr = x*x + y*y;
                if (sqr==num)
                {
                    array[i]=x;
                    array[i+1]=y;
                    i=i+2;
                    count++;
                }
                x--;
            }
            y++;
            x=round(sqrt(num/2));
        }
    }
    if(count*2>=4)
    {
        cout<<”The value of a, b, c and d: \n”;
        for(i=0;i<count*2;i=i+2)
            cout<<”=>”<<array[i]<<” and
”<<array[i+1]<<endl;

        int temp1 = abs(array[0] - array[2]);
        int temp2 = abs(array[1] - array[3]);

        int k = gcd(temp1,temp2);
        int l = temp1/k;
        int m = temp2/k;

        int n = (array[0] + array[2])/m;
        if(gcd(l,m)==1)
            cout<<”Program is working fine!
\n\n”;
        else
            cout<<”Check for errors! \n\n”;
        int factor1 = pow((0.5*k),2) +
pow((0.5*n),2);    //Find factors.
        int factor2 = pow(l,2) + pow(m,2);
        cout<<"The factors for the number given
are: "<<factor1<< " * "<< factor2<<"\n";
    }
    else
        cout<<"\nEuler Factorization method is
not applicable for the number given. \n";
}
```

- <u>Dixon's Method of Factorization</u>

  Implementation of this algorithm is done in Java –

```
import java.util.*;
import java.lang.*;
//importing libraries
import java.io.*;

import java.util.Scanner;
class Exe
{
    public static void main(String args[])
    {
    Scanner ob=new Scanner(System.in);
    System.out.print("Enter value of N:  ");
    double n=ob.nextDouble();
    System.out.println("n ---------------------
---------------------");
    dixon(n);
    }
    public static double  gcd(double a,double b)
    {
        if(b==0)
        return a;
        else if(b>a)
        return gcd(b,a);
        else
        return gcd(b,a%b);
    }
    public static int checkprime(double n)
    {
    int f=0;
      for(int i=2;i<=n/2;i++)
      {
        if(n%i==0)
        {
            f=1;
            break;
        }
              }
             if(f==1)
                return 0;
            else
                return 1;
    }
    public static void dixon(double n)
    {
        int a,d,b1,d1;
        double m,x,p,q;
        m=Math.sqrt(n);
        double c=Math.floor(m);
        int f1=0;
        for(double f=c+1;f<=n;f++)
        {
            double s,m1;
            s=Math.sqrt((f*f)%n);
            m1=Math.floor(s);
            if((s-m1)==0)
            {
                if(s!=(f%n))
                {
                p=f+s;
                q=f-s;
                b1=checkprime(p);
                d1=checkprime(q);
                if(b1==1)
                {
                double z=gcd(q,n);
                int b=checkprime(z);
                if(b==1){
                System.out.println(" ----------
-----------------------------------------");
                System.out.println("  THE TWO
FACTORS ARE    "+z+"  and    "+p);
                System.out.println(" ----------
------------------------------------------");
                }
            }
```

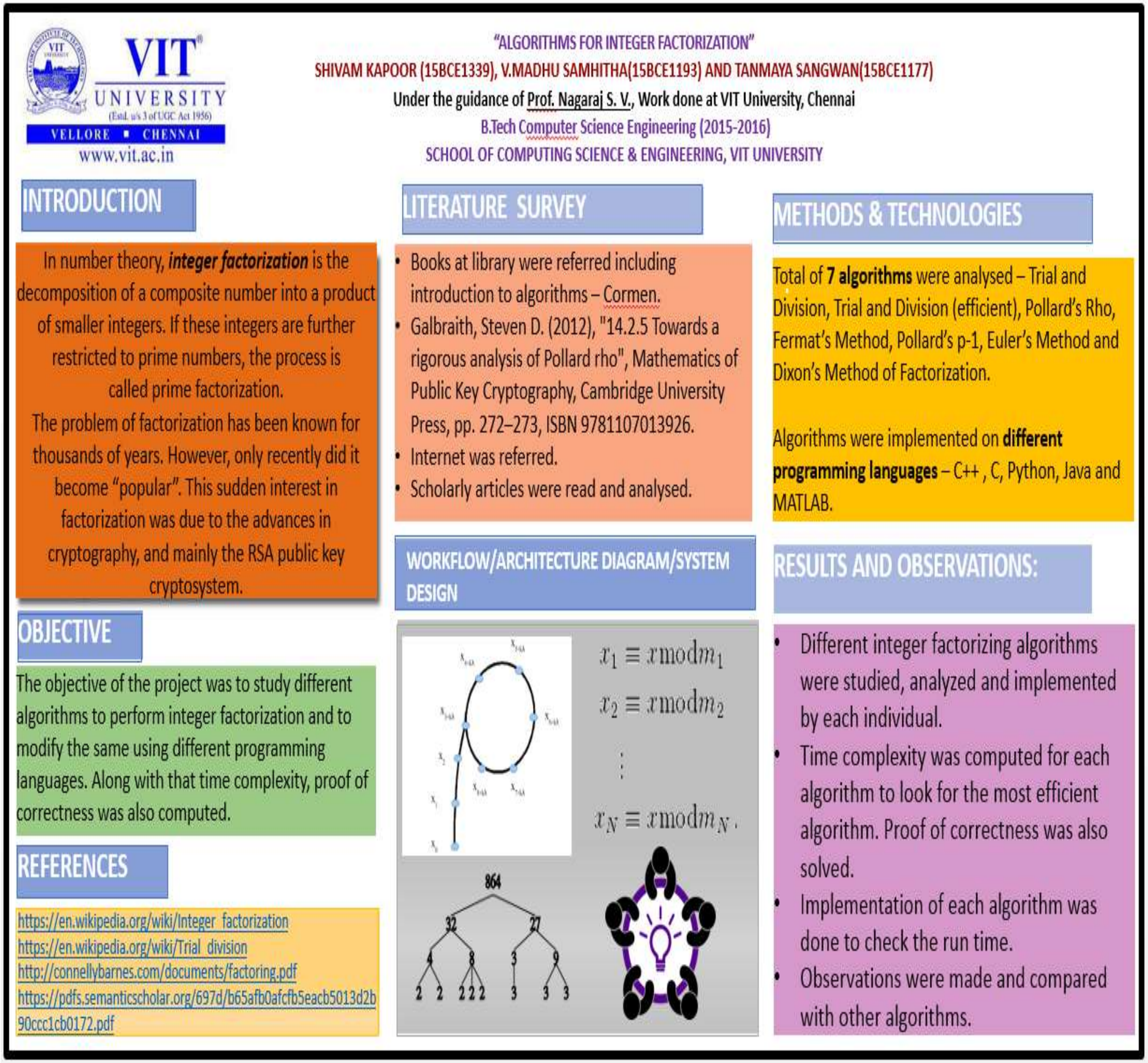
```
                else if(d1==1)
                {
                double z=gcd(p,n);
                int b=checkprime(z);
                if(b==1){
                System.out.println("-----------
-------------------------------");
                System.out.println("    THE TWO
FACTORS ARE     "+z+"       and    "+q);
                System.out.println("-----------
-------------------------------");}
                }
                else
                System.out.println("-----------
--- NEITHER OF THE FACTORS ARE PRIME---------
" );
                f1=1;
                break;
                }
            }
                if(f1==1)
                break;
        }
    }
}
```

- <u>KID - RSA</u>

  Implementation of this algorithm is done in MATLAB –

```
clc
clear all
syms a b a1 b1 Msg Enc
Msg=input('Please Enter your message: ');
a=input('Please Enter the value of a: ');
b=input('Please Enter the value of b: ');
a1=input('Please Enter the value of a1: ');
b1=input('Please Enter the value of b1: ');
M = (a*b) - 1;
e = (a1*M) + a;
d = (b1*M) + b;
n = ((e*d)-1)/M;
Encryption = mod(Msg * e, n)
Enc = input('Please Enter the Encrypted value: ');
Decryption = mod(Enc * d, n)
```

POSTER PRESENTATION



"ALGORITHMS FOR INTEGER FACTORIZATION"

SHIVAM KAPOOR (15BCE1339), V.MADHU SAMHITHA(15BCE1193) AND TANMAYA SANGWAN(15BCE1177)

Under the guidance of Prof. Nagaraj S. V., Work done at VIT University, Chennai

B.Tech Computer Science Engineering (2015-2016)

SCHOOL OF COMPUTING SCIENCE & ENGINEERING, VIT UNIVERSITY

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

INTRODUCTION

In number theory, ***integer factorization*** is the decomposition of a composite number into a product of smaller integers. If these integers are further restricted to prime numbers, the process is called prime factorization.

The problem of factorization has been known for thousands of years. However, only recently did it become "popular". This sudden interest in factorization was due to the advances in cryptography, and mainly the RSA public key cryptosystem.

OBJECTIVE

The objective of the project was to study different algorithms to perform integer factorization and to modify the same using different programming languages. Along with that time complexity, proof of correctness was also computed.

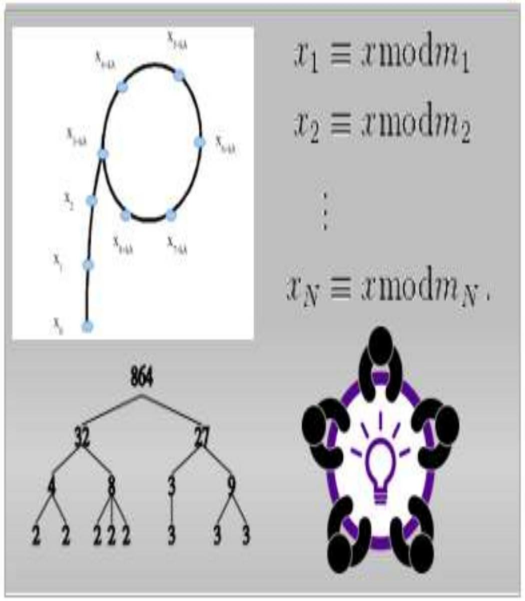
REFERENCES

https://en.wikipedia.org/wiki/Integer_factorization
https://en.wikipedia.org/wiki/Trial_division
http://connellybarnes.com/documents/factoring.pdf
https://pdfs.semanticscholar.org/697d/b65afb0afcfb5eacb5013d2b90ccc1cb0172.pdf

LITERATURE SURVEY

- Books at library were referred including introduction to algorithms – Cormen.
- Galbraith, Steven D. (2012), "14.2.5 Towards a rigorous analysis of Pollard rho", Mathematics of Public Key Cryptography, Cambridge University Press, pp. 272–273, ISBN 9781107013926.
- Internet was referred.
- Scholarly articles were read and analysed.

WORKFLOW/ARCHITECTURE DIAGRAM/SYSTEM DESIGN



$$x_1 \equiv x \bmod m_1$$
$$x_2 \equiv x \bmod m_2$$
$$\vdots$$
$$x_N \equiv x \bmod m_N.$$

METHODS & TECHNOLOGIES

Total of **7 algorithms** were analysed – Trial and Division, Trial and Division (efficient), Pollard's Rho, Fermat's Method, Pollard's p-1, Euler's Method and Dixon's Method of Factorization.

Algorithms were implemented on **different programming languages** – C++ , C, Python, Java and MATLAB.

RESULTS AND OBSERVATIONS:

- Different integer factorizing algorithms were studied, analyzed and implemented by each individual.
- Time complexity was computed for each algorithm to look for the most efficient algorithm. Proof of correctness was also solved.
- Implementation of each algorithm was done to check the run time.
- Observations were made and compared with other algorithms.

REFERENCES

- https://en.wikipedia.org/wiki/Integer_factorization
- https://en.wikipedia.org/wiki/Trial_division
- https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm
- https://en.wikipedia.org/wiki/Fermat%27s_factorization_method
- https://en.wikipedia.org/wiki/Euler%27s_factorization_method
- http://mathworld.wolfram.com/EulersFactorizationMethod.html
- http://mathworld.wolfram.com/PollardRhoFactorizationMethod.html
- http://www.csh.rit.edu/~pat/math/quickies/rho/
- Galbraith, Steven D. (2012), "14.2.5 Towards a rigorous analysis of Pollard rho", Mathematics of Public Key Cryptography, *Cambridge University Press, pp. 272–273,* ISBN 9781107013926.
- http://connellybarnes.com/documents/factoring.pdf
- https://en.wikipedia.org/wiki/Dixon%27s_factorization_method
- https://pdfs.semanticscholar.org/697d/b65afb0afcfb5eacb5013d2b90ccc1cb0172.pdf

ALGORITHMS FOR INTEGER FACTORIZATION

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