INTEGER FACTORIZATION

By

- **1.Diptarka Biswas (Roll no. 042812)**
- **2.Abhiman Biswas (Roll no. 042813)**
- **3.Bishan Samaddar(Roll no. 042816)**

B.E.(Information Technology)

JADAVPUR UNIVERSITY

UNDER THE SUPERVISION OF

Prof. Utpal Kumar Ray

Acknowledgement

We would like to thank a lot of people without whose help and encouragement this term paper would not have been completed.

First and foremost, We thank our guide Mr. Utpal Ray. His suggestions and guidance helped us immensely in understanding the subject.

We would also like to thank our friends for the constant support and help they have provided me all the time.

Diptarka Biswas

Abhiman Biswas

Bishan Samaddar

B.E.(Information Technology)

JADAVPUR UNIVERSITY

Abstract of the thesis

From the time of Fermat, the mathematics community is trying to find newer ways to improve integer factorization. From Fermat's Factorization method to Quadratic Sieve and Number Field Sieve, the advances in factorization techniques has come a long way. But it still remains as one of the toughest challenges in Number Theory. It takes months to factorize a 512 bit integer using parallel supercomputers for number crunching. And it has become a daunting task, as nowadays 1024 and 2048 bit keys are used in encryption systems(in RSA).

The security of RSA depends on the ability of factorizing huge numbers. Here we present a seminal idea of factorizing integers using innovative techniques from concrete and Vedic Mathematics. Using simple examples and informal discussions this article surveys the key ideas and major advances of the last quarter century in integer factorization.

Organization of the thesis

This paper gives a brief survey of integer factorization algorithms. We offer several motivations for the factorization of large integers. A number of factoring algorithms are then explained, mainly a factorization method which is properly defined based on the trick of factoring the difference of two squares to find the factor of a big integer.

- ***Chapter 1*** is a brief description about different factorization method including the Elliptic curve factorization [ECM] & General Number Field Sieve Algorithm (GNFS).
- ***Chapter 2*** discusses about a survey on factoring large numbers.
- ***Chapter 3*** discusses about the implementation of The Sequential Quadratic Field Sieve Factorization algorithm with big integer library (GMP library).
- ***Chapter 4*** discusses about possible approach for parallelized integer factorization method.
- ***Chapter 5*** discusses about GNU Multiple Precision Arithmetic Library.
- ***Chapter 6*** gives sufficient information about the various requirements to execute the program.

- ***Chapter 7*** is the conclusion of the project work, also includes further research work.

CHAPTER 1

DIFFERENT TYPES OF FACTORIZATION METHODS

There are several Integer Factorization Methods. We only give the description of few modern methods. They are as follows

- 1. ***Trial Division***
- 2. ***Pollard's "rho" Algorithm***
- 3. ***Pollard's "p-1" Algorithm***
- 4. ***Elliptic Curve Factorization***
- 5. ***Quadratic Field Sieve Algorithm***
- 6. ***General Number Field Sieve Algorithm***

1.1 Trial Division:

"Trial division" tries to factor a positive integer n by checking whether n is divisible by 2, checking whether n is divisible by 3, checking whether n is divisible by 4, checking whether n is divisible by 5, and so on. Trial division stops after checking whether n is divisible by y; here y is a positive integer chosen by the user.

If n is not divisible by 2, 3, 4, . . . , d-1, but turns out to be divisible by d, then trial division prints d as output and recursively attempts to factor n=d. The divisions by 2 , 3, 4, . . . , d-1 can be skipped

divisions by inside the recursion: n=d is not divisible by 2, 3, 4, . . . , d-1.

1.1.1 Algorithm Description:

We just try potential divisors d=2, 3, 4…until one of the following occurs-

1. d > √N, in which case N is prime; or
2. d < N and d | N, in which case d is nontrivial prime divisor of N; or
3. d exceeds some preassigned bound B < √N, in which case all we can say is that any prime factor p of N satisfies p > B.

1.1.2 Example:

Consider the problem of factoring n = 314159265. Trial division with y = 10 perform the following computations:

- 314159265 is not divisible by 2;
- 314159265 is divisible by 3, so replace it with 314159265/3 = 104719755 and print 3;
- 104719755 is divisible by 3, so replace it with 104719755/3 = 34906585 and print 3 again;
- 34906585 is not divisible by 3;
- 34906585 is not divisible by 4;
- 34906585 is divisible by 5, so replace it with 6981317 and print 5;
- 6981317 is not divisible by 5;
- 6981317 is not divisible by 6;
- 6981317 is not divisible by 7;
- 6981317 is not divisible by 8;
- 6981317 is not divisible by 9;
- 6981317 is not divisible by 10.

This computation has revealed that 314159265 is 3 times 3 times 5 times 6981317;the factorization of 6981317 is unknown.

1.2 The Pollard's "rho" Algorithm:

The rho method was introduced by Pollard in [107]. The name "rho" comes from the shape of the Greek letter ρ; the shape is often helpful in visualizing the graph with edges $\rho_i \bmod p \quad \rho_{i+1} \bmod p$, where p is a prime.

Define $\rho_1, \rho_2, \rho_3, \ldots$ by the recursion $\rho_{i+1} = \rho_i^2 + 10$, starting from $\rho_1 = 1$. The "rho method" tries to find a nontrivial factor of n by computing

$$\gcd\{n, (\rho_2 - \rho_1)(\rho_4 - \rho_2)(\rho_6 - \rho_3) \ldots .(\rho_{2z} - \rho_z)\};$$

where z is a parameter chosen by the user.

There's nothing special about the number 10. The "randomized rho method" replaces 10 with a uniform random element of {3, 4, . . . , n-3}.

1.2.1 Algorithm Description:

- 1.Pollard's "rho" algorithm uses an iteration of the form

  $X_{i+1} = f(X_i) \bmod N, \qquad i \geq 0,$

- 2. In practice a quadratic polynomial

  $f(X) = X^2 + a$

  is used ($a \neq 0, -2 \bmod N$).

- 3.The probability that $X_0, X_1, \ldots, X_k$ are all distinct mod p is approximately $(1-1/p)(1-2/p)\ldots(1-k/p) \sim \exp(-k^2/(2p))$, and if $X_0, X_1, \ldots, X_k$ are not all distinct mod p then $j \leq k$.

1.2.2 Example:

Let the number to be factored is 77.

Let $X_{n+1} = X_n^2 + 1$

We are calculating the value of x_i using mod 77.

i	x_i	Gcd (N, x_i-$x_{i/2}$)
1	1	
2	2	1
3	5	
4	26	1
5	61	
6	26	7

So, a factor of 77 is 7.

1.3 The Pollard's "p-1" Algorithm:

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.

Let n be a composite integer with prime factor p. By Fermat's little theorem, we know that

$a^{K(p-1)} \equiv 1 \pmod{p}$ for all K, and for all a co prime to p

If a number x is congruent to 1 modulo a factor of n, then the gcd (x-1,n) will be divisible by that factor.

The idea is to make the exponent a large multiple of p-1 by making it a number with very many prime factors; generally, we take the product of all prime powers less than some limit B. Start with a random x, and repeatedly replace it by $x^w \mod n$ as w runs through those prime powers. Check at each stage, or once at the end if you prefer, whether (x-1, n) is not equal to 1.

1.3.1 Algorithm Description:

Pollard's "p – 1" algorithm [24, 27] is based on Fermat's theorem

$$a^{p-1} = 1 \bmod p$$

for 0<a<p, p prime. Suppose that p is a prime factor of N and that E is a multiple of p-1. Then from Fermat's theorem,

$$p \mid \mathrm{GCD}(a^E - 1, N)$$

and GCD(a^E – 1, N) gives us a factor (possibly trivial) of N.

Since p is not known in advance , the algorithm supposes that all prime power factors of p-1 are bounded above by some arbitrarily chosen number m. Then, taking E as a product of prime powers q^e, $q^e \le m$, we obtain a multiple of p-1. If p-1 has a prime factor greater than m, then the algorithm will generally fail to give a nontrivial factor of N.

Because E is very large, it is not actually computed. Instead, a^E mod N is computed via a sequence of computations

$$a \leftarrow a^{q^e} \bmod N.$$

The worked involved in such an exponentiation is $O(\log(q^e))$ multiplications mod N, so the total work involved is O(m) multiplications mod N, and the time required is $O(m.(\log N)^2)$.

1.3.2 Example:

If y = 10 and z = 10 then E(y,z) = $2^3 \cdot 3^2 \cdot 5 \cdot 7 = 8 \cdot 9 \cdot 5 \cdot 7 = 2520$. The p – 1 method tries to find a nontrivial factor of n by computing gcd{ n,2^{2520} – 1}. To understand why this is effective, look at how many primes divide 2^{2520} - 1: 3, 5, 7, 11, 13, 17, 19, 29, 31, 37, 41, 43, 61, 71, 73, 109, 113, 127, 151, 181, 211, 241, 281, 331, 337, 421, 433, 631, 1009, 1321, 1429, 2521, 3361, 5419, 14449, 21169, 23311, 29191, 38737, 54001, 61681, 86171, 92737, etc.

Here y and z are parameters chosen by the user, and E(y, z) is a product of powers of primes ≤ z, the powers being chosen to be as large as possible while still being ≤ y; in other words, E(y, z) is the least common multiple of all z-smooth prime powers ≤ y.

1.4 Elliptic Curve Factorization:

The Lenstra elliptic curve factorization or the elliptic curve factorization method (ECM) is a fast, sub-exponential running time algorithm for integer factorization which employs elliptic curves

Iteratively applying a group function to a series of points starting on a random point in a group defined by an elliptic curve modulo the number we are factorizing we will eventually find a generator for the subgroup we iterate over. Using the order of this subgroup, we can determine a factor of *n*.

For general purpose factoring, ECM is the third-fastest known factoring method. The second fastest is the multiple polynomial quadratic sieve and the fastest is the general number field sieve; both are probabilistic algorithms, in the sense there is no guarantee that they will return a non trivial factor in the expected running time; there is an exponentially small probability they will take an exponential amount of time

1.4.1 Algorithm Description:

The Lenstra elliptic curve factorization method to find a factor of the given number *n* works as follows:

- Pick a random elliptic curve over $\mathbf{Z}/n\mathbf{Z}$, given by an equation of the form $y^2 = x^3 + ax + b$, and a non-trivial point *A* on it. The group law on this curve is given by rational functions; divisions of residue classes modulo *n* can be performed using the Euclidean algorithm. If in the remaining part of the algorithm a non-zero element of $\mathbf{Z}/n\mathbf{Z}$ is encountered which fails to be invertible, we get in effect a factorization of *n*.

- Compute *eA* in the group of ($\mathbf{Z}/n\mathbf{Z}$)-valued points of the elliptic curve, where *e* is product of small primes raised to small powers, as in the $p - 1$ algorithm. This can be done one prime at a time, thus efficiently.

- Hopefully, *eA* is a zero element of the elliptic curve group in $\mathbf{Z}_p$ but not in $\mathbf{Z}_q$, for two prime divisors *p* and *q* of *n* — as in the *p* − 1 method, it is unlikely that both groups will have an order which is a divisor of *e*. Then we can find a factor of *n* by finding the greatest common divisor of the *x*-coordinate of *eA* and *n*, since this coordinate will be zero in $\mathbf{Z}_p$.

- If it does not work, we can try again with some other curve and starting point.

The complexity depends on the size of the factor and can be represented by O ($e^{(1 + o(1))\sqrt{(\ln p \ln \ln p)}}$), where *p* is the smallest factor of *n*.

1.4.2 Example:

Let the number to be factored be 77.
Let the elliptic curve be: $y^2 = x^3 + x + 7$
Here a = 1, b = 7.
$4a^3 + 27b^2 = 1327 \neq 0$
Let a point A on the curve have coordinates (1, 3).
Let M = 2. So, M! = 2.
Therefore M!A = (7, 70)
GCD (7, 77) = 7.
So, 7 is a factor of 77.

1.5 Quadratic Field Sieve Algorithm:

Quadratic Sieve

- A method for factoring large numbers.
- In practice, the second fastest factoring algorithm, exceeded only by the number field sieve.
- Running time dependent only on n, the number to be factored.
  - ❖ Running time: $O(n) = e^{\sqrt{\log(n)\log(\log(n))}}$
  - ❖ Running time for NFS: $e^{1.9223((\ln(n))1/3\ (\ln(\ln(n)))2/3\ )}$

Congruence of Squares

Consider $x^2 \equiv y^2 \pmod{n}$, where $x \neq \pm y \pmod{n}$.

$$x^2 \equiv y^2 \pmod{n}$$

$$\text{Or, } x^2 - y^2 \equiv 0 \pmod{n}$$

$$\text{Or, } (x + y)(x - y) \equiv 0 \pmod{n}$$

Thus we see that (x + y) and (x - y) must each have a factor of n.

gcd(x + y, n) and gcd(x - y, n) will give us these factors.

The problem: Finding such x and y.

Finding Congruence

We want to find:

$x^2_i \equiv y_i \pmod{n}$ for i = 1, 2, ..., r and $y_1 y_2 \ldots y_r$ is a perfect square.

Then $(x_1 x_2 \ldots x_r)^2 \equiv y_1 y_2 \ldots y_r = c^2 \pmod{n}$.

Further problem: How to find a set of yi by a method other than trial and error.

Finding Congruence with Vector Addition

We want to first select a 'factor base' of primes. For example, select {2, 3, 5, 7} as our factor base.

Find numbers which can be decomposed into a product of these prime factors only.

For example, $150 = 2^1 \cdot 3^1 \cdot 5^2 \cdot 7^0$ and $295 = 2^1 \cdot 3^1 \cdot 5^0 \cdot 7^2$

Now we can write these numbers as 'exponent vectors:'

$$150 = < 1, 1, 2, 0 > \text{ and } 295 = < 1, 1, 0, 2 >$$

The entries in each of these vectors represent the power of each prime factor in the integer base.

Now we can say that:

$150 \cdot 295 = <1, 1, 2, 0> + <1, 1, 0, 2> = <2, 2, 2, 2> = 2^2 \cdot 3^2 \cdot 5^2 \cdot 7^2 = 44100$.

We have found a square with the form $2^2 \cdot 3^2 \cdot 5^2 \cdot 7^2 = (2 \cdot 3 \cdot 5 \cdot 7)^2$ using vector addition.

So we can use vector addition to find squares from numbers in the form of our 'factor base.'

Furthermore, we only care if the vector components are even. So we can consider all vectors mod 2 and discover the zero vector.

Theorem: Given k elements of a vector, a linear dependency will occur among k + 1 different vectors.

Translation: Using this method, we are assured of finding a square with k + 1 vectors.

Thus the basic process of discovering a square should be apparent:

- Choose a factor base of k elements.
- Find numbers (up to k + 1) that decompose into the elements of the factor base.
- Write each number as an exponent vector (mod 2).
- Find the proper addition of vectors to produce the zero vector (mod 2).
- Find the square as shown above.

QS Application of Vector Addition

In practice, the factor base for the quadratic sieve method is chosen as:

- Include (-1).
- Include 2.
- Include a number of odd primes pi where n is a quadratic residue of pi. In other words, $(n/p_i) = 1$.

Generally, choose an upper bound B and include all acceptable pi less than B.

A smaller size for the factor base will mean that less matrix calculations will be required. However, this is offset by the difficulty of finding numbers which can be represented by the smaller factor base.

Oppositely, a larger factor base will mean more matrix calculations are required, but it is quicker to find numbers to use.

It's a tradeoff.

Now let $f(x) = x^2 - n$.

We will start testing x which are 'near' $\sqrt{n}$. Specifically, $x = \sqrt{n} \pm j$, where $j \in Z$, preferably small.

We are only interested in $f(x_i)$ which factor in our chosen base, and throw out all others.

All $f(x_i)$ that factor within our chosen base are stored as the appropriate vector, called $v(x_i)$.

In theory, we may need k +1 such f(x) to be assured of a solution, but in practice, a lesser number will often do.

Using methods of linear algebra (such as Gaussian Elimination), find $v(x_1) + v(x_2) + \ldots + v(x_s) = < 0, 0, \ldots, 0 >$.

Thus we have $(x_1x_2 \ldots\ldots x_s)^2 \equiv f(x_1)f(x_2) \ldots\ldots f(x_s) = y^2 \pmod{n}$.

The 'Sieve'

Now that we have a method, how do we check the factorization of each f(xi)?

$$f(x) = x^2 - n$$
$$f(x + kp) = (x + kp)^2 - n$$
$$f(x + kp) = x^2 - n + 2xkp + (kp)^2$$
$$f(x + kp) = f(x) + 2xkp + (kp)^2 = f(x) \pmod{p}$$

Thus we have that if $p|f(x)$, $p|f(x + kp)$ for any integer k.

This creates a sieve for locating factors. For example, if $7|f(81)$, then we also know that $7|f(88)$, $7|f(95)$, and so forth.

This allows us to determine one factor of one $f(x_i)$ and know many other $f(x_i)$ which also have that factor.

The standard method to use the sieve is to just start dividing out factors as they get found to make finding the remaining factors easier.

peaking from a theoretical perspective, this method is the most accurate. However, it requires a large amount of memory.

Tying it all together

Methodology for factoring a number using the quadratic sieve:

- Choose a bound B.
- Choose a factor base $\{-1, 2, p_2, \ldots p_r\}$, where $(n/p_i) = 1$ and $p_i < B$.
- Calculate $f(x) = x^2 - n$ for x 'near' $\sqrt{n}$.
- Using the sieve appropriately, find a 'sufficient number' of x for which f(x) can be decomposed into the factor base.
- Store these f(xi) as vectors in the factor base, mod 2.
- Using vector addition, discover a linear combination that produces the zero vector.
- Find the congruence $(x_1 \ldots\ldots x_s)^2 \equiv f(x_1) \ldots\ldots f(x_s) = y^2 \pmod{n}$.

1.5.1 Example:

Let n = 9487, and we will choose a bound of B = 30.

Using this bound of 30, the factor base is < -1, 2, 3, 7, 11, 13, 17, 19, 29 >.

We now start calculating f(x) until a sufficient number is reached; we are assured an answer at 10. We test x = √9475 ± j. In other words, x = 97, 98, 96, 99, 95, 100, 94, etc.

x	f(x)	-1	2	3	7	11	13	17	19	29
81	-2926	1	1	0	1	1	0	0	1	0
84	-2431	1	0	0	0	1	1	1	0	0
85	-2262	1	1	1	0	0	1	0	0	1
89	-1566	1	1	1	0	0	0	0	0	1
95	-462	1	1	1	1	1	0	0	0	0
97	-78	1	1	1	0	0	1	0	0	0
98	117	0	0	0	0	0	1	0	0	0
100	513	0	0	1	0	0	0	0	1	0
101	714	0	1	1	1	0	0	1	0	0
103	1122	0	1	1	0	1	0	1	0	0

For our first try, we can calculate that v(85) + v(89) + v(98) = < 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 >.

Thus we have $(85 \cdot 89 \cdot 98)^2 \equiv (2 \cdot 3^2 \cdot 13 \cdot 29)^2 \pmod{9487}$.

Unfortunately, when we reduce both sides, we find that $7413702 \equiv 203582 \pmod{9487}$.Thus we have a useless solution.

Looking again, we can see that v(81) + v(95) + v(100) is also equal to the zero vector.

Thus we have $(81 \cdot 95 \cdot 100)^2 \equiv (2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 19)^2 \pmod{9487}$.

Reducing mod 9487, we get $10532 \equiv 7360 \pmod{9487}$.

gcd(1053 + 7360, 9487) = 179. Dividing 9487 by 179, we get our second factor, 53.

1.6 General Number Field Sieve Algorithm:

The General Number Field Sieve algorithm is the fastest known method for factoring large integers. Research and development of this algorithm within

the past five years has facilitated factorizations of integers that were once speculated to require thousands of years of supercomputer time to accomplish. While this method has many unexplored features that merit further research, the complexity of the algorithm prevents almost anyone but an expert from investigating its behavior.

In mathematics, the ***general number field sieve*** (GNFS) is the most efficient classical algorithm known for factoring integers larger than 100 digits. Heuristically, its complexity for factoring an integer n is of the form

$$O\left(e^{(c+o(1))(\log n)^{1/3}(\log\log n)^{2/3}}\right) = L_n[1/3, c]$$

(in O and L notations) for a constant c which depends on the complexity measure and on the variant of the algorithm[1]. It is a generalization of the special number field sieve: while the latter can only factor numbers of a certain special form, the general number field sieve can factor any number (apart from prime powers, but this is a minor issue). When the term *number field sieve* (NFS) is used without qualification, it refers to the general number field sieve.

1.6.1 Algorithm Description:

- We choose two polynomials *f(x)* and *g(x)* of small degrees *d* and *e*, which have integer coefficients, which are irreducible over the rationales, and which, when interpreted mod *n*, have a common root *m*. An optimal strategy for choosing these polynomials is not known; one simple method is to pick a degree *d* for a polynomial, consider the expansion of *n* in base *m* (allowing digits between -*m* and *m*) for a number of different *m* of order $n^{1/d}$, and pick *f(x)* as the polynomial with the smallest coefficients and *g(x)* as *x*-*m*.
- A better method was suggested by Murphy and Brent [2]; they introduce a two-part score for polynomials, based on the presence of roots modulo small primes and on the average value that the polynomial takes over the sieving area.
- The best reported results [2] were achieved by the method of Thorsten Kleinjung [3], which allows *g(x)* = *ax*+*b*, and searches over *a*

composed of small prime factors congruent to 1 modulo *2d* and over leading coefficients of *f* which are divisible by 60.

- Now, we consider the number field rings **Z[r1]** and **Z[r2]**, where **r1** and **r2** are roots of the polynomials *f* and *g*, and look for values *a* and *b* such that $r = b^d \cdot f(a/b)$ and $s = b^e \cdot g(a/b)$ are smooth relative to the chosen basis of primes. If *a* and *b* are small, then *r* and *s* will be too (but at least of order of *m*), and we have a better chance for them to be smooth at the same time. The current best-known approach for this search is lattice sieving; to get acceptable yields, it is necessary to use a large factor base, and generally sieving with large primes is used.
- Having enough such pairs, using Gaussian elimination, we can get products of certain *r* and of the corresponding *s* to be squares at the same time. We need a slightly stronger condition—that they are norms of squares in our number fields, but we can get that condition by this method too. Each *r* is a norm of *a*- **r1***b* and hence we get that the product of the corresponding factors *a*- **r1***b* is a square in **Z[r1]**, with a "square root" which can be determined (as a product of known factors in **Z[r1]**)—it will typically be represented as an irrational algebraic number. Similarly, we get that the product of the factors *a*- **r2***b* is a square in **Z[r2]**, with a "square root" which we can also compute.
- Since *m* is a root of both *f* and *g* mod *n*, there are homomorphisms from the rings **Z[r1]** and **Z[r2]** to the ring **Z/nZ**, which map **r1** and **r2** to *m*, and these homomorphisms will map each "square root" (typically not represented as a rational number) into its integer representative. Now the product of the factors *a*-*m**b* mod *n* can be obtained as a square in two ways—one for each homomorphism. Thus, we get two numbers *x* and *y*, with x^2-y^2 divisible by *n* and again with probability at least one half we get a factor of *n* by finding the greatest common divisor of *n* and *x*-*y*.

CHAPTER 2

A SURVEY ON FACTORING LARGE NUMBERS

2.1 Introduction:

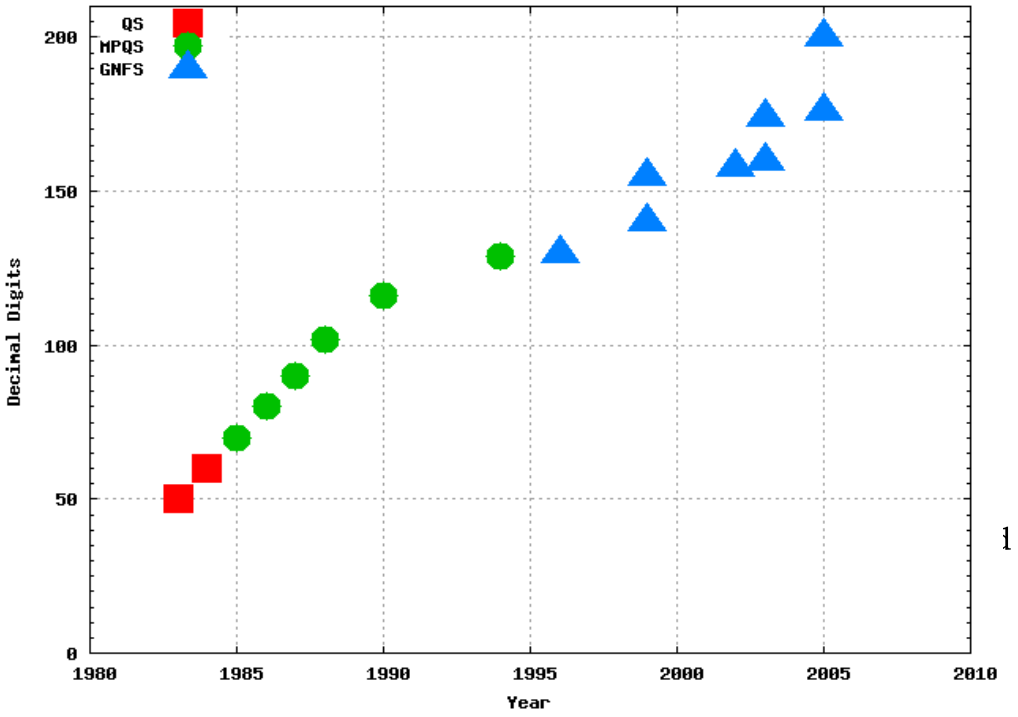
- Factoring a number means representing it as the product of smaller numbers.
- It is difficult to factor a large number.
- Some cryptosystems are based on the difficulty of the factoring integer problem.
- It measures the security of the cryptosystems to factor large numbers in short time.

2.2 Comparison of some factorization methods:

Factorization Methods	**For all Numbers**	**Technique Used**
Trial Division	Can not factorize	if "n mod i = 0" for i = 2,3,4,…
Pollard (p-1)	Can not factorize	$(a^{k!} -1) \bmod n$
Pollard Rho	Can not factorize	Periodic sequences
Quadratic field Sieve	Can factorize	Congruent squares
Elliptic Curve Multiplication	Can Factorize	Elliptic curves
General Number Field Sieve	Can Factorize	Polynomial definition step Sieving step Matrix solving step Making square root step

2.3 Calculation Records:

● Factoring Records:



- It will take 1 year to sieve 1024bit length number
- Not in practice yet

- Quantum Computing
  - Shor's algorithm may run very fast
  - Quantum computer is not in practice

CHAPTER 3

SEQUENTIAL INTEGER FACTORIZATION METHOD

3.1Introduction:

While Fermat's method is much faster than trial division, when it comes to the real world of factoring, for example factoring an RSA modulus several hundred digits long, the purely iterative method of Fermat is too slow. Several other methods have been presented, such as the Elliptic Curve Method discovered by H. Lenstra in 1987 and a pair of probabilistic methods by Pollard in the mid 70's, the p –1 method and the ρ method. The fastest algorithms, however, utilize the same trick as Fermat, examples of which are the Continued Fraction Method, the Quadratic Sieve (and it variants), and the Number Field Sieve (and

its variants). The exception to this is the Elliptic Curve Method, which runs almost as fast as the Quadratic Sieve. The remainder of this paper focuses on the Quadratic Sieve Method.

Mathematicians have been attempting to find better and faster ways to factor composite numbers since the beginning of time. Initially this involved dividing a number by larger and larger primes until you had the factorization. Trial division was not improved upon until Fermat applied the factorization of the difference of two squares: $a^2-b^2 = (a-b)(a+b)$. In his method, we begin with the number to be factored: n. We find the smallest square larger than n, and test to see if the difference is square. If so, then we can apply the trick of factoring the difference of two squares to find the factors of n. If the difference is not a perfect square, then we find the next largest square, and repeat the process.

In this chapter we have discussed about the implementation of sequential integer factorization method using quadratic field sieve algorithm.

- Algorithm used : Quadratic Field Sieve
- Language used : C Programming language
- Platform used : Red Hat Linux
- Library used : GNU Multiple Precision Library (GMP)

GNU Multiple Precision Library (GMP) which is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers have been used. There is no practical limit to the precision except the ones implied by the available memory in the machine.

3.3 Algorithm:

Step-1:

Let the number to be factored be N. We choose smoothness bound B where B is calculated by the equation

$$B=2\sqrt{N} - 1.$$

Step-2:

Calculate the factor base denoting a set of prime numbers which are less than B, will control both the length of the vectors and the number of vectors needed. The factor base only needs to contain such primes p for which Legendre symbol (N/p)=1.

Note: (a/p) = 0, if p|a

(a/p) = 1, if a is a square modulo p – that is to say there exists an integer k such that $k^2 \equiv a \pmod p$, or in other words a is a quadratic residue modulo p.

(a/p) =-1, if a is not a square modulo p, or in other words a is not a quadratic residue modulo p.

Step-3:

Calculate the nearest square root of N which is denoted by q and also calculate the value of x_i where $x_i = q \pm i$. Calculate $y_i = x_i^2 - N$.

Note: We should not select such y_i which has a factor not existing in the factor base.

Step-4:

Produce a matrix by the selected y_i and its number of occurrence of the factor base. The occurrence is then calculated by mod 2.

Step-5:

Transpose the matrix. Let the matrix is called A.

Step-6:

Reduce the matrix by row reduction method and apply mod 2 operation.
We have to find a non-zero vector X such that A.X=0 using gaussian elimination method.

Step-7:

According to the position value 1 in the non zero vector find $Y=y_n * y_m$ where n & m positions in the non zero vectors. Similarly find $X=x_n * y_n$ for the same position n & m, where X & Y will satisfy $X^2 \equiv Y^2 \bmod N$.

Step-8:

Calculate the GCD of {(X-Y), N} which is a factor of N.

Step-9:
Repeat this process for modified N.

3.3 An Example of the above Algorithm:

Let the number to be factored be 77.
Let B = 15.
The factor base is {-1, 2, 13}
So, π(B) = 3.
The integer nearest the square root of 77 is 8.
The first few B-smooth values of $y_i = x_i^2 - 77$ are:
The smoothness pairs are (8,-13),(9,4),(5,-52),(-5,-52)

If we put the mod 2 numbers in a matrix it will be:

[1 0 1]
[0 0 0]
[1 0 1]
[1 0 1]

Let the transpose of the above matrix be A.

$$A = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

We want to find nonzero vector X such that AX = 0. The problem can be solved using row reduction (Gaussian elimination). We row reduce the matrix, and then assign values to the free variables in a way that gives us a nonzero solution. The other variables will be determined by these values and the matrix.

After gaussian solution
we get [1 0 1 0], corresponding to -13 * -52 = 26^2.
Let y = 26.

Let $x = x_1 * x_3 = 8 * 5 = 40$.
So, we can see that $40^2 \equiv 26^2 \pmod{77}$
Therefore GCD (x - y, N) = GCD (14, 77) = 7.
So, 7 is a factor of 77.

If we get [1 0 0 1], corresponding to $-13 * -52 = 26^2$
Let y = 26
Let $x = x_1 * x_4 = 8 * -5 = -40 \equiv 37 \pmod{77}$
So, we can see that $37^2 \equiv 26^2 \pmod{77}$
Therefore GCD (x - y, N) = GCD (11, 77) = 11.
So, 11 is a factor of 77.

3.4 Analysis of the program code:

To handle big numbers, in the program "gmp.h" has been included in the header file. All programs using GMP should include this file.The numbers n is given as input during the run time of the program. The variables of the program are 'n' , 'root', 'factor_size', 'sieve_length', 'i', 'j', 'tmp', '*factor_ptr', '*sp_ptr', '**mtx' in the function 'main'. In the function 'main' we initialize n (number to be factored), factor_size(number of factors in the factor base), sieve_length (limit of checking prime factors).The different functions that we have used are

- void factor_base(int n,int * factor_ptr,int factor_size);
- int is_smooth(int n,int *factor_ptr,int factor_size,int index);
- void matrix(int ** mtr,int *factor_ptr, int factor_size,int *sieve_ptr ,int sp_size);
- void transpose(int ** mtx, int factor_size);
- int gsolve(int ** mtxt,int factor_size);
- int gcd(int m,int n);

void factor_base(int n,int * factor_ptr,int factor_size);

This function will create the set of prime factors according to the values of factor_size,number to be factored. Smoothness value is calculated by the given by $2*\sqrt{n}-1$.In this function we are checking the legendre value. Depending on the legendre values we select the prime values into the factor base.

int is_smooth(int n,int *factor_ptr,int factor_size,int index);

This function is used for checking smoothness.

void matrix(int ** mtr,int *factor_ptr, int factor_size,int *sieve_ptr ,int sp_size);

The matrix is produced in the mod2 form of exponents of different factors of the factor base for each and every y, Where $y = x^2 - N$.

void transpose(int ** mtx, int factor_size);

This function is used to transpose the matrix.

int gsolve(int ** mtxt,int factor_size);

This function involves the implementation of the Gaussian Solution method. With the help of Gaussian solution we calculate the non zero vectors from the produced matrix.

int gcd(int m,int n);

The function gcd (int m, int n); is used to find the gcd of (x-y) and N.And this function will produce us the ultimate factor of the given integer input N.
In the function main() if we do not find enough smooth pairs i.e., if base count is less than factor_size+1, then an error will be produced i.e, not enough smooth pairs found. The memory is allocated dynamically for factor base and the matrix.

CHAPTER 4

POSSIBLE APPROACH FOR PARALLELIZED INTEGER FACTORIZATION METHOD

Many computing tasks involve heavy mathematical calculations, or analyzing large amounts of data. These operations can take a long time to

complete using only one computer. Networks such as the Internet provide many computers with the ability to communicate with each other. Parallel or distributed computing takes advantage of these networked computers by arranging them to work together on a problem, thereby reducing the time needed to obtain the solution.

The drawback to using a network of computers to solve a problem is the time wasted in communicating between the various hosts. The application of distributed computing techniques to a space environment or to use over a satellite network would therefore be limited by the amount of time needed to send data across the network, which would typically take much longer than on a terrestrial network.

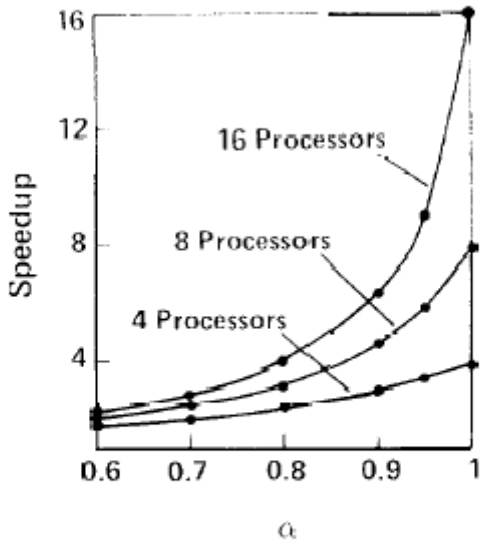
This experiment shows how much faster a large job can be performed by adding more computers to the task, what role communications time plays in the total execution time, and the impact a long-delay network has on a distributed computing system.

When designing parallel algorithms we hope that an algorithm which requires time T_1 on a computer with one processor can be implemented to run in time $T_P \sim T_1/P$ on a computer with P independent processors. This is not always the case, since it may be impossible to use all P processors effectively. However, it is true for many integer factorization algorithms, provided P is not too large. The *speedup* of a parallel algorithm is $S = T_1/T_P$. We aim for a linear speedup, i.e. $S = \theta(P)$.

The key issue in the parallel processing of a single application is the speedup achieved, especially its dependence on the number of processors used. We

define speedup *(S)* as the factor by which the execution time for the application changes: that is,

$$S = \frac{\text{execution time for one processor}}{\text{execution time for } p \text{ processors}}$$



Speedup as a function of parallelism (a) and number of processors.

CHAPTER 5

GNU MULTIPLE PRECISION ARITHMETIC LIBRARY

5.1 Introduction to GNU MP:

GNU MP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types.

Many applications use just a few hundred bits of precision; but some applications may need thousands or even millions of bits. GMP is designed to give good performance for both, by choosing algorithms based on the sizes of the operands, and by carefully keeping the overhead at a minimum.

The speed of GMP is achieved by using full words as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code for the most common inner loops for many different CPUs, and by a general emphasis on speed (as opposed to simplicity or elegance).

5.2 Installing GMP:

GMP is a free library and it is available for download from the site www.swox.com/gmp

It has an autoconf/automake/libtool based configuration system. On a Unix-like system a basic build can be done with

```
$./configure
$make
```

Some self-tests can be run with

```
$make check
```

And you can install (under /usr/local by default) with

```
$make install
```

This is the standard installation procedure. But, I follow the following installation procedure to install the GMP in our computer.

First, download the GMP package gmp-4.2.1.tar.gz from the above link. This is the zip file. So, we unzip the package using the following command –

```
$gunzip –c gmp-4.2.1.tar.gz | tar xf -
```

After that the unzip folder gmp-4.2.1 save in your home directory then we use touch command to update all the file in the folder gmp-4.2.1

```
$touch gmp-4.2.1/*
```

Then, we enter the folder gmp-4.2.1 using command –

```
$cd gmp-4.2.1
```

On a Unix-like system a basic build can be done with

```
$./configure
$make
```

Some self-tests can be run with

```
$make check
```

5.3 GMP Basics:

Using functions, macros, data types, etc. not documented in this manual is strongly discouraged. If you do so your application is guaranteed to be incompatible with future versions of GMP.

5.3.1 Headers and Libraries:

All declarations needed to use GMP are collected in the include file gmp.h. It is designed to work with both C and C++ compilers.

```
#include <gmp.h>
```

Prototypes for GMP functions with FILE * parameters are only provided if <stdio.h> is included too.

```
#include <stdio.h>
#include <gmp.h>
```

All programs using GMP must link against the 'libgmp' library. On a typical Unix-like system this can be done with '-lgmp', for example

```
gcc myprogram.c -lgmp
```

5.3.2 Nomenclature and Types:]

In GMP library integer usually is defined as a multiple precision integer, The C data type for such integers is mpz_t. Here are some examples of how to declare such integers:

```
mpz_t sum;
struct foo { mpz_t x, y; };
mpz_t vec[20];
```

Rational number means a multiple precision fraction. The C data type for these fractions is mpq_t. For example:

```
mpq_t quotient;
```

Floating point number or ***Float*** for short, is an arbitrary precision mantissa with a limited precision exponent. The C data type for such objects is mpf_t. For example:

```
mpf_t fp;
```

The floating point functions accept and return exponents in the C type mp_exp_t. Currently this is usually a long, but on some systems it's an int for efficiency.

A ***limb*** means the part of a multi-precision number that fits in a single machine word. (We chose this word because a limb of the human body is analogous to a digit, only larger, and containing several digits). Normally a limb is 32 or 64 bits. The C data type for a limb is mp_limb_t.

Counts of limbs are represented in the C type mp_size_t. Currently this is normally a long, but on some systems it's an int for efficiency.

Random state means an algorithm selection and current state data. The C data type for such objects is gmp_randstate_t. For example:

```
gmp_randstate_t rstate;
```

Also, in general unsigned long is used for bit counts and ranges, and size_t is used for byte or character counts.

5.4Function Classes:

There are six classes of functions in the GMP library:

- **1.** Functions for signed ***integer arithmetic***, with names beginning with mpz_. The associated type is mpz_t. There are about 150 functions in this class.

- **2.** Functions for ***rational number arithmetic***, with names beginning with mpq_. The associated type is mpq_t. There are about 40 functions in this class, but the integer functions can be used for arithmetic on the numerator and denominator separately.

- **3.** Functions for ***floating-point arithmetic***, with names beginning with mpf_. The associated type is mpf_t. There are about 60 functions is this class.

- **4.** Functions compatible with ***Berkeley MP***, such as itom, madd, and mult. The associated type is MINT.

- **5.** Fast ***low-level functions*** that operate on natural numbers. These are used by the functions in the preceding groups, and you can also call them directly from very time-critical user programs. These functions' names begin with mpn_. The associated type is array of mp_limb_t. There are about 30 (hard-to-use) functions in this class.

- **6.** ***Miscellaneous functions***. Functions for setting up custom allocation and functions for generating random numbers.

5.4.1 Integer Function:

Initialization Functions

Initialization is done by calling the function mpz_init.

void mpz_init (mpz_t integer)

Initialize integer and set its value to 0.

void mpz_clear(mpz_t integer)

Free the space occupied by the integer

For example

{

```
mpz_t integ;
mpz_init (integ);
……..
mpz_clear (integ);

}
```

Assignment Function

These functions assign new values to already initialized integers.

```
void mpz_set (mpz_t rop, mpz_t op)
void mpz_set_ui (mpz_t rop, unsigned long int op)
void mpz_set_si (mpz_t rop, signed long int op)
```

Set the value of rop from op.

```
Int mpz_set_str (mpz_t rop, char *str, int base)
```

Set the value of ***rop*** from ***str***, a null-terminated C string in base ***base***. The base may vary from 2 to 62, or if base is 0, then the leading characters are used: 0x and 0X for hexadecimal, 0b and 0B for binary, 0 for octal, or decimal otherwise.

Combined Initialization and Assignment Functions

GMP provides a parallel series of initialize-and-set functions which initialize the output and then store the value there.

```
void mpz_init_set (mpz_t rop, mpz_t op)

void mpz_init_set (mpz_t rop, mpz_t op)

void mpz_init_set_si (mpz_t rop, signed long int op)
```

Initialize rop with limb space and set the initial numeric value from op.

```
int mpz_init_set_str (mpz_t rop, char *str, int base)
```

Here is an example :

```
{
        mpz_t pie;
        mpz_init_set_str(pie,“3141592653589793238462643383279
        502884",10);
        …
        mpz_clear (pie);
}
```

Arithmetic Functions

void mpz_add (mpz_t rop ,mpz_t op1, mpz_t op2)

void mpz_add_ui (mpz_t rop, mpz_t op1, unsigned long int op2)

Set ***rop*** to ***op1 + op2***.

void mpz_sub (mpz_t rop ,mpz_t op1, mpz_t op2)

void mpz_sub_ui (mpz_t rop, mpz_t op1, unsigned long int op2)

Set ***rop*** to ***op1 - op2.***

void mpz_mul (mpz_t rop ,mpz_t op1, mpz_t op2)

void mpz_mul_ui (mpz_t rop, mpz_t op1, unsigned long int op2)

Set ***rop*** to ***op1*** time's ***op2***.

Division Functions

Division is undefined if the divisor is zero. Passing a zero divisor to the division or modulo functions will cause an intentional division by zero.

void mpz_mod (mpz_t r, mpz_t n, mpz_t d)

unsigned long int mpz_mod_ui (mpz_t r, mpz_t n, unsigned long int d)

Set ***r*** to ***n mod d***. The sign of the divisor is ignored; the result is always non-negative.

void mpz_divexact (mpz_t q, mpz_t n, mpz_t d)

Set ***q*** to ***n/d.***

Comparison Function

int mpz_cmp (mpz_t op1, mpz_t op2)

int mpz_cmp_ui (mpz_t op1, unsigned long int op2)

Compare ***op1*** and ***op2***. Return a positive value if ***op1*** > ***op2***, zero if ***op1*** = ***op2***, or a negative value if ***op1*** < ***op2***.

5.6 GMP calls used in our program:

We use several GMP calls in our program. We describe those commands in the given below –

- mpz_t
- mpz_init

- mpz_set_str
- mpz_sqrt
- mpz_add_ui
- mpz_fdiv_q_ui
- mpz_set_ui
- mpz_cmp
- mpz_mod
- mpz_cmp_ui
- mpz_divexact
- mpz_mul_ui
- mpz_sub_ui
- gmp_printf
- gmp_scanf
- mpz_clear

CHAPTER 6

VARIOUS REQUIREMENTS TO RUN THE PROGRAM

6.1 System Requirements:

Basically System Requirements are the combination of both Hardware & Software requirements. We need a sufficient environment to run the program properly. Needed s/w and h/w requirements are given below.

6.1.1 Hardware Requirements:

- Operating System : Red Hat Linux 9.0
  Mandrake Linux 11
  Fedora Core 2.6.9-1.667 smp

- Machine Configuration : Intel® Pentium ® 4 CPU 3 GHz
  With HT technology (2 logical processors)
  3.00 GHz, 1 GB of RAM.
  Cache 1 MB L2,
  52x CD-ROM Drive,
  A minimum of 28 kbps and Modem

6.1.2 Software Requirements:

- The program has been implemented in GNU C language.
- GMP multiple Precision Library Version 4.2.2 has been used to handle large numbers.

CHAPTER 7

7.1 Conclusion:

Currently, Number Field Sieve is being used for prime factorization. Most of the algorithms are derived from one or more parent algorithm. Therefore, we can say that only 3 or 4 standard algorithms are present for prime factorization. Rest of the algorithms is an improvement over these root factorization methods.

This algorithm is purely an academic interest to us. Just as the quote by Gauss and like all other mathematicians, we were romancing with numbers and hit upon something interesting and innovative.

The security of e-commerce applications based on public key cryptography such as RSA depends on the difficulty of factorizing large integers. Given the progress in the development of new factorization methods, the increase in the computational power of personal computers, and the emergence of well-organized group of users such as distributed net, organization should deploy public key cryptography with key length long enough to make the factorization attack difficult.

7.2 Further research work:

Integer factorization is an extremely challenging discipline. The scopes for improvements are huge which also gives us a new insight into the world of primes and prime factorization. I want to learn more about the mathematical theory underlying the integer factorization algorithms and then I want to implement general number field sieve algorithm using modern computer technology in small computer network.

References:

1. www.wikipedia.org

2. www.planetmath.org

3. http://www-fs.informatik.uni-tuebingen.de/~reinhard/krypto/Factor/QuadAlgE.html

4. http://www.math.umbc.edu/~campbell/NumbThy/Class/BasicNumbThy.html

5. http://www.csh.rit.edu/~pat/math/quickies/rho/

6. http://www-math.mit.edu/18.310/22.-Factoring-II-Elliptic-Curves.pdf

7. http://www.certicom.com

8. http://blogs.msdn.com/devdev/archive/2006/06/19/637332.aspx

9. Parallel algorithms for integer factorization by Richard P. Brent.

10. A Beginner's Guide to the General Number Field Sieve by Michael Case

11. Recent Progress and Prospects for Integer Factorization Algorithms
- by Richard P. Brent.

12. The Quadratic Sieve Factoring Algorithm by Eric Landquist.

13. Note on Integer Factoring Methods-I by Nelson A. Carella.

14. FACTOR: An integer factorization program for the IBM pc by Richard P. Brent.

15. Adaptive Scheduling for Task Farming with Grid Middleware
Henri Casanova, MyungHo Kim, James S. Plank, Jack J. Dongarra.