Integer Factorisation Problem & Modern Factoring Algorithms

Vishisht Priyadarshi
180123053
vishisht@iitg.ac.in

Sidharth Bankupalle
180123047
bankupal@iitg.ac.in

May 20, 2020

Abstract

"The problem of distinguishing prime numbers from composite numbers, and of resolving the latter into their prime factors, is known to be one of the most important and useful in arithmetic."

The Integer Factorization problem is defined as follows: **Given a composite integer N, find a non-trivial factor e of N, i.e. find e such that e divides N.**

When the numbers are sufficiently large, no efficient, non-quantum integer factorization algorithm is known.The most straightforward method of factoring is trial division, which, essentially, checks for every prime number p, such that $p \leq \sqrt{n}$, if p divides N. However, trial division may take more than $O\left(\sqrt{N}\right)$ bit operations, which makes it extremely inefficient for the numbers of interest.

Currently there is no known algorithm for answering the question "Does integer n have a factor less than integer s?" in a number of steps that is $O\left(P\left(n\right)\right)$, where n is the number of digits in N, and $P\left(n\right)$ is a polynomial function. Moreover, no one has proved that such an algorithm exists, or does not exist

In this project, we aim to do an analysis of the modern Factoring Algorithms and their constraints.

Introduction

Integer factoring is a well-known number theory problem with wide applications in complexity and cryptography. In this project, we will discuss these factoring algorithms in detail - making way through all the intricate details and their time complexity analysis:
1. Fermat's Factorisation Method
2. Pollard's p-1 Factorisation Algorithm
3. Pollard's Rho Algorithm
4. Linear Sieve

1. Fermat's Factorisation Method

PROBLEM: The Fermat's Factorisation method can be applied to arbitrary odd n to find a divisor pair that are relatively close together, if such a pair exists.

MAIN IDEA:

Consider n to be an odd number(without loss of generality). If n has a non-trivial factors, then n can be written as $n = (x + y)(x - y)$ for some numbers $x, y \in \mathbb{N}$ s.t. $x + y$, $x - y$ is not equal to 1 and n.

To implement this concept, we define a variable a with initial value of $x^2 - n$ where x is initialised with a value of $\lfloor\sqrt{n}\rfloor$. Now we iterate over a loop until we found that a is a perfect square. We update the value of x over each iteration by adding odd numbers to it in a sequential way, starting from $2x + 1$. That is, initially $a = x^2 - n$, then after first iteration it becomes $(x + 1)^2 - n$ since $(x + 1)^2 = x^2 + 2x + 1$. With each iteration, a takes the values $(x + 1)^2 - n$, $(x + 2)^2 - n$ and so on until loop terminates. At the termination of the loop, we would have $a = x'^2 - n$ where a can be written as $a = y^2$. So, we have $n = x'^2 - y^2$ and hence we have successfully factored n.

<u>ALGORITHM:</u>

```
$x \leftarrow \lfloor\sqrt{n}\rfloor$
$t \leftarrow 2x + 1$
$a \leftarrow x^2 - n$
while($a$ is not a square of some $k \in \mathbb{N}$)
{
    $a \leftarrow a + t$
    $t \leftarrow t + 2$
}
$x \leftarrow (t - 1)/2$
$y \leftarrow \sqrt{a}$
return $x + y$ and $x - y$ as factors of $n$
```

<u>TIME COMPLEXITY:</u>
Let the odd composite number n be written as $n = ab$. Then at the termination $a = x - y$ and $b = x + y$. Now to count the number of iterations, we can take help of t. Initially, t will begin with $2x + 1$. So, roughly t can be considered to be $2\sqrt{n}$ and at the termination $t = 2x + 1 = 1 + a + b$.
Hence the number of iterations = $\frac{1}{2}(1 + a + b - 2\sqrt{n}) = \frac{1}{2}\left(1 + a + \frac{n}{a} - 2\sqrt{n}\right)$
$$= \frac{1}{2}\left(1 + \frac{(\sqrt{n} - a)^2}{a}\right)$$

As it is evident that the algorithm performs poorly than Trial Division in some of the cases. It works well only when a is very close to $\sqrt{n}$.

2. Pollard's p-1 Factorisation Algorithm

<u>PROBLEM:</u> Suppose we have a positive integer n such that it can be expressed as product of 2 primes numbers, i.e. $n = pq$, where, p and q are primes. Our aim is to find out

p and q.

MAIN IDEA:

p is prime $\Rightarrow$ p - 1 is composite
Let $p-1 = a_1^{i_1} a_2^{i_2} \ldots a_k^{i_k}$,where $a_1, a_2, \ldots, a_k$ are primes.
Consider a number L s.t. $L \geq \max\left\{a_1^{i_1}, a_2^{i_2}, \ldots, a_k^{i_k}\right\}$
Now L! = L.(L-1).(L-2)...1 must be divisible by p-1, since all prime powers of (p-1) exist in the terms of L! at least once.

By Fermat's Little Theorem, for any arbitrary a: $a^{p-1} \equiv 1\,(mod\;p)$
Now since $L! \equiv 0\,(mod\;\;(p-1)) \Rightarrow a^{L!} \equiv 1\,(mod\;p) \Rightarrow a^{L!} - 1 \equiv 0\,(mod\;p)$
$\Rightarrow$ p divides $a^{L!} - 1$.
$\therefore$ $\gcd\left(a^{L!} - 1, n\right) = p$ or n
We can compute this GCD. If it is not equal to n, we have found a prime factor of n, i.e., p.
But if $\gcd\left(a^{L!} - 1, n\right) = n \Rightarrow$ n divides $a^{L!} - 1 \Rightarrow$ q divides $a^{L!} - 1 \Rightarrow a^{L!} \equiv 1\,(mod\;q)$.
Now consider $L! = j\,(q-1) + k$, j, k $\in \mathbb{Z}$ and k $\in$ [0, q-1]
$\Rightarrow a^{L!} \equiv a^k\,(mod\;q) \Rightarrow \therefore a^k \equiv 1\,(mod\;q)$

Case (i) k = 0 : We restart the process with a different random value of a.
Case (ii) k $\neq$ 0 : $a^k \equiv 1\,(mod\;q)$ holds true always until a is not divisible by q. But in no way, we can know before-hand if this is the case since p, q are unknown to us. So the best possible way is to restart the process with a random value of a.

ALGORITHM:

```
a ← random()
for j ← 2 to BOUND(B)
{
    a ← a^j mod n
    d ← gcd(a-1, n)
    if(1 < d < n)
        return d
    else if(d = n)
        RESTART ALGORITHM WITH A NEW VALUE OF a
}
```

TERMINATION:

As we compute $a^{L!}$, and take gcd we have two cases:
Case (i): If d > 1, we found the gcd and loop terminates.
Case (ii): If d = n, we go back and restart the algorithm with a new random value of a. It is guaranteed that we will run into some value of a which is multiple of one of the prime factors of n. In that case, gcd(a, n) > 1. Say, p divides a, and p also divides $a^{L!} - 1$
$\Rightarrow \gcd\left(a^{L!} - 1,\; n\right) = p$. Hence the algorithm terminates.

TIME COMPLEXITY:
Lets discuss time complexity for a single iteration with a fixed value of a.
There are $O\left(B\right)$ modular exponentiations, and each requiring $O\left(logB\right)$ multiplications.
GCD can be computed in O(log n) time.
$\therefore$ Time Complexity $= O(B.logB.logn + logn)$

3. Pollard's Rho Algorithm

PROBLEM: Suppose we have a positive integer n such that it can be expressed as product of 2 primes numbers, i.e. $n = pq$, where, p and q are primes. Our aim is to find out p and q.

MAIN IDEA:
Pollard's rho algorithm iterates a simple polynomial map (which should not be a permutation polynomial) and produces a non-trivial divisor of n. The map generates a pseudo-random sequence of numbers.
Pollard's rho algorithm uses an iteration of the form: $x_{i+1} = f\left(x_i\right) mod\ n \quad , i \neq 0$
where n is the number to be factored, x_0 is a random starting value and f is a polynomial with integer coefficients. In practice, $f\left(x\right) = x^2 + a$ is chosen (a $\neq$ 0, -2 since they are fixed points for $f\left(x\right)$).

Now the algorithm proceeds as follows:
STEP 1. : Compute $d = gcd\left(x - x', n\right), \ \ x' = f\left(x\right)$
STEP 2. : If $1 < d < n$, d is a proper divisor of n.
If d = 1, replace x by $f\left(x\right)$, and x' by $f\left(f\left(x'\right)\right)$
If d = n, algorithm has to be re-initialised, since we encountered a failure.

In short, we are looking for two distinct values $x_i, x_j \in \mathbb{Z}_n$ s.t. $gcd\left(x_i - x_j, n\right) > 1$. This is reasoned out by the fact that if $gcd\left(x_i - x_j, n\right) > 1 \Rightarrow \ gcd\left(x_i - x_j, n\right)$ is a factor of n, which is either p or q.
But whether we will get $gcd\left(x_i - x_j, n\right) > 1$ and that too in reasonable number of steps is answered by the Birthday Paradox. In fact, Pollard's Rho Algorithm can be seen as a re-formulation of Birthday Paradox Problem, which we will discuss now:

Now say we pick k numbers $x_1, x_2, \ldots, x_k$ uniformly at random in $\mathbb{Z}_n$. Suppose that these k numbers are distinct. Now let's say that there exists some $1 \neq i < j \neq k$ s.t.
$x_i = x_j\left(mod\, p\right)$. Then $p \mid x_i - x_j$, and since $p \mid n$ also , we have $p \mid gcd\left(x_i - x_j, n\right)$.
Moreover since $x_i, x_j \in \mathbb{Z}_n$ and they are distinct
$\Rightarrow gcd\left(x_i - x_j, n\right) < n \Rightarrow \therefore gcd\left(x_i - x_j, n\right)$ provides a non-trivial factor of n.
But we need to find an upper-bound on size of k such that it is guaranteed that if we compute $gcd\left(x_i - x_j, n\right)$ for all pairs of $1 \leq i < j \leq k$, we find a non-trivial factor.

Now,

$$x_i \equiv x_j \,(mod\; p) \tag{1}$$

Lets calculate the probability that equation (1) does not hold, i.e., $(x_1, x_2, \ldots, x_k)\, mod\; p$ are all different.

$$\mathbb{P} = \left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right)\ldots\left(1 - \frac{k-1}{n}\right) \leq e^{-\frac{1}{n}}.e^{-\frac{2}{n}}\ldots e^{-\frac{k-1}{n}} = e^{\frac{-k\,(k-1)}{2n}} \sim e^{\frac{-k^2}{2n}} \tag{2}$$

Since $p \leq \sqrt{n}$, $k = n^{\frac{1}{4}}$ is suitable since it gives a higher probability that (1) holds
Now we use the pseudo-random generator function to generate the k numbers in sequence and search for the pair i, j s.t. $i < j$ and $x_i \equiv x_j\,(mod\; p)$. But checking all the $\binom{k}{2}$ pairs will result in a lot of GCD computations, thereby resulting in time complexity which is of the order $\sqrt{n}$.

An optimisation that we can make here is to reduce the number of GCD computations by using Floyd's Cycle Detection Algorithm. We can easily show that:
$x_i \equiv x_j\,(mod\; p) \rightarrow x_{i+1} \equiv x_{j+1}\,(mod\; p)$ which we can generalise as:
$x_{i+\delta} \equiv x_{j+\delta}\,(mod\; p)\,\forall\;\; \delta 0$.
Eventually the sequence runs into a cycle. So our task reduces to finding $x_i \equiv x_j\,(mod\; p)$ with $i < j$ and i as small as possible.
Now it is an established fact that there will be a pair $i < j < C_o\sqrt{p}$ $\quad$ (C_o is a constant) s.t. $x_i \equiv x_j\,(mod\; p)$ (based on empirical calculations). So our algorithm finds out the factor of n in $O\left(n^{1/4}\right)$ time.

ALGORITHM:

```
INITIALISE POLYNOMIAL MAP $f$
$x \leftarrow x_o$
$x' \leftarrow f(x)\,(mod\; n)$
$p \leftarrow$ gcd($x - x', n$)
while($p$ = 1)
{
    $x \leftarrow x_i$
    $x' \leftarrow x_{2i}$
    $x \leftarrow f(x)\,(mod\; n)$
    $x' \leftarrow f(x')\,(mod\; n)$
    $x' \leftarrow f(x')\,(mod\; n)$
    $p \leftarrow$ gcd($x - x', n$)
}
if($p$ = $n$)
    return "$n$ is prime"
```

```
else
    return $p$
```

TIME COMPLEXITY:
As discussed in the previous section, the Pollard's Rho algorithm is able to find a factor p of n with an expected number of $\Theta\left(\sqrt{p}\right)$ of arithmetic operations, that is, about an order of $\Theta\left(n^{1/4}\right)$.

4. Linear Sieve

PROBLEM: Given a number n, find all prime numbers in a segment $[2, n]$ in linear time complexity

MAIN IDEA:
The sieve of Eratosthenes is probably the simplest way to pick out all the primes in a given range from 2 to n but it has time complexity $O\left(n \log\log n\right)$. The algorithm described below is interesting by its simplicity: it isn't any more complex than the classic sieve of Eratosthenes but has linear complexity.
Our goal is to calculate minimum prime factor $lp\left[i\right]$ for every number i in the segment $[2, n]$. Besides, we need to store the list of all the found prime numbers - let's call it $pr\left[\right]$. We'll initialize the values $lp\left[i\right]$ with zeros, which means that we assume all numbers are prime. During the algorithm execution this array will be filled gradually. Now we'll go through the numbers from 2 to n. We have two cases for the current number i:

Case 1 : $lp\left[i\right] = 0$ - This means that i is prime, i.e. we haven't found any smaller factors for it. Hence, we assign $lp\left[i\right] = i$ and add i to the end of the list $pr\left[\right]$.
Case 1 : $lp\left[i\right] \neq 0$ - This means that i is composite, and its minimum prime factor is $lp\left[i\right]$.

In both cases we update values of $lp\left[\right]$ for the numbers that are divisible by i. However, our goal is to learn to do so as to set a value $lp\left[\right]$ at most once for every number. We can do it as follows:

Let's consider numbers $x_j = i \cdot p_j$, where p_j are all prime numbers less than or equal to $lp\left[i\right]$ (this is why we need to store the list of all prime numbers).
We'll set a new value $lp\left[x_j\right] = p_j$ for all numbers of this form.

ALGORITHM:

```
Enter a natural number n
Initialise array lp[n]
Initialise list pr
for i <- 2 to n
{
    if(lp[i] = 0)
    {
        lp[i] <- i
        push back i to pr
    }
    for j <- 1 to size(pr)
    {
        if(pr[j] <= lp[i] AND (i*pr[j]) <= n)
        {
            lp[i*pr[j]] = pr[j]
        }
    }
}
return lp[]
```

<u>TIME & SPACE COMPLEXITY:</u>

Although the running time of $O(n)$ is better than $O(n \log\log n)$ of the classic sieve of Eratosthenes, the difference between them is not so big. In practice that means just double difference in speed, and the optimized versions of the sieve run as fast as the algorithm given here. Considering the memory requirements of this algorithm - an array $lp[]$ of length n, and an array of $pr[]$ of length $\frac{n}{\ln n}$, this algorithm requires 32 times the memory needed for the classic sieve. However, its usefulness is that this algorithm calculates an array $lp[]$, which allows us to find factorization of any number in the segment $[2;n]$ in the time of the size order of this factorization. Moreover, using just one extra array will allow us to avoid divisions when looking for factorization. Knowing the factorizations of all numbers is very useful for some tasks, and this algorithm is one of the few which allow to find them in linear time.

Conclusion

We are interested in factoring, because it is an example of an algorithmic problem on which there has been well-documented progress. As well as their inherent interest and applicability to other areas of mathematics, advances in public key cryptography have lent algorithms for integer factorisation a lot of practical importance. Despite much progress in the development of efficient algorithms, our knowledge of the complexity of factorisation is inadequate. We would like to find a polynomial time factorisation algorithm or else prove that one does not exist. Until a polynomial time algorithm is found or a quantum computer capable of running Shor's algorithm(a polynomial time quantum computer algorithm) is built, large factorisations will remain an interesting challenge.

The general number field sieve (GNFS) is the most efficient classical algorithm known for

factoring integers larger than 10^{100}. It is a randomized algorithm and heuristically, we get an expected complexity of $O\left(e^{\sqrt{\frac{64}{9}}(\log N)^{\frac{1}{3}}(\log\log N)^{\frac{2}{3}}}\right)$.
GNFS has been to able factor numbers up to 576 bits long. However, it seems quite improbable that a PC-based implementation of the algorithm can do much better. That is why, other, non-algorithmic methods, have been proposed to improve the efficiency of the existing algorithms.

Some of these includes ideas of Distributed Computing, use of special hardwares(like Sieve Devices) and use of Quantum Computers (Shor's Algorithm).

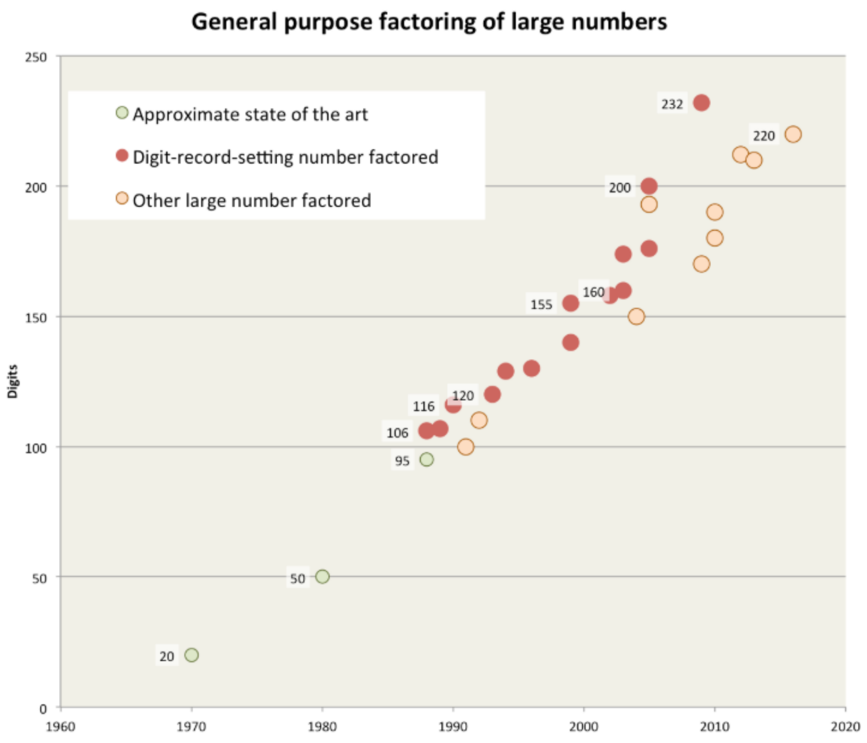

Figure 1: Figure 1: Size of numbers (in decimal digits) that could be factored over recent history. Green 'approximate state of the art' points do not necessarily represent specific numbers or the very largest that could be at that time—they are qualitative estimates. The other points represent specific large numbers being factored, either as the first number of that size ever to be factored (red) or not (orange). Dates are accurate to the year. Some points are annotated with decimal digit size, for ease of reading.

References

1. Arjen K. Lenstra, *General purpose integer factoring*
2. Kostas Bimpikis and Ragesh Jaiswal, *Modern Factoring Algorithms*
3. Richard P. Brent, *An improved Monte Carlo Factorization Algorithm*, (1980)
4. Peter L. Montgomery, *Speeding the Pollard and Elliptic Curve Methods of Factorization*, *Mathematics of Computation*, Volume 48, Issue 177 (Jan.,1987), 243-264
5. Eric Bach ,*Toward a Theory of Pollard's Rho Method*, (1991)

6. David Gries & Jayadev Misra, *A Linear Sieve Algorithm for Finding Prime Numbers*
7. Anne-Sophie Charest, *Pollard's p-1 and Lenstra's factoring algorithms*, (2005)
8. Sounak Gupta and Goutam Paul, *Revisiting Fermat's Factorization for the RSA Modulus* (2009)
9. Robert Erra and Christophe Grenier, *The Fermat factorization method revisited*, $(3othJune2009)$
10. Soud K. Mohamed, *Analysis of Pollard's Rho Factoring Method*, (2019)
11. Samuel S. Wagstaff, Jr., *The Joy of Factoring*, Student mathematical library Volume 68, *Pg - 119 to 142*
12. Cristina-Loredana Duta et al., *Framework for evaluation and comparison of integer factorization algorithms*, 2016 SAI Computing Conference (2016)