Integer Factorisation Problem & Modern Factoring Algorithms

Vishisht Priyadarshi & Sidharth Bankupalle

Indian Institute of Technology Guwahati

May 2020

Content

- Introduction
- Factorisation Algorithms:
  - Fermats's Factorisation Algorithm
  - Pollards p-1 Algorithm
  - Pollards Rho Algorithm
  - Linear Sieve
- Conclusion & Future Prospects
- References

Introduction

Why do we need to care about Integer Factorisation

- 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

Introduction (Contd.)

- We are interested in factoring, because it is an example of an algorithmic problem on which there has been well-documented progress. Such examples should inform our expectations for algorithmic problems in general (including problems in AI).
- 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
- In this project, we aim to do an analysis of the modern Factoring Algorithms and their constraints.

1. Fermat's Factorisation Method

Algorithmic Perspective

- 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 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.

Algorithm

Pseudo Code

```
INPUT : Odd natural number, n
x <- Sqrt(n)
t <- 2x + 1
a <- x^2 - n
while(a is not a square of some k in N)
{
    a <- a + t
    t <- t + 2
}
x <- (t - 1)/2
y <- Sqrt(a)
return (x+y) and (x-y) as factors of n
```

Time Complexity

Analysis

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 $t = 2x + 1 = 1 + a + b$.

$$\begin{aligned}\text{Hence the number of iterations} &= \frac{1}{2}\left(1 + a + b - 2\sqrt{n}\right) \\ &= \frac{1}{2}\left(1 + a + \frac{n}{a} - 2\sqrt{n}\right) \\ &= \frac{1}{2}\left(1 + \frac{\left(\sqrt{n} - a\right)^2}{a}\right)\end{aligned}$$

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 $\sqrt{n}$.

Example

Factorising n = 527

x	t = 2x+1	x^2	a = $x^2 - n$
22	45	484	-43
23	47	529	2
24	49	576	49

Hence $527 = 24^2 - 7^2 = (24 - 7)(24 + 7)$

2. Pollard's p-1 Factorisation Algorithm

Algorithmic Perspective

- Pollard's p - 1 Method is based on Fermat's Little Theorem which says that $a^{p-1} \equiv 1 \, (mod \ p)$ when p is a prime which does not divide a.
- Therefore $a^L \equiv 1 \, (mod \ p)$ for any multiple L of $p-1$. If also $p|n$, then p divides $gcd\left(a^L - 1, n\right)$. Of course, we cannot compute a^L mod p because p is an unknown prime factor of n.
- However, we can compute a^L mod n. Pollard's idea is to let L have many divisors of the form $p1$ and thus try many potential prime factors p of n at once.

Deeper Insights

- 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]
  $\Rightarrow a^{L!} \equiv a^k\,(mod\ q) \Rightarrow \therefore a^k \equiv 1\,(mod\ q)$

Deeper Insights (Contd.)

- 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.
- 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.

Algorithm

Pseudo Code

```
INPUT : Natural number, n
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
}
```

Time Complexity

Key Points

- There are $O(B)$ modular exponentiations, and each requiring $O(logB)$ multiplications.
- GCD can be computed in O(log n) time.
- $\therefore$ Time Complexity $= O(B.logB.logn + logn)$

Example

Factorising n = 10001

Let the random number be 2, i.e., $a = 2$.

j	a = $a^j \% n$	d = gcd$(a-1, n)$
1	2	1
2	4	1
3	64	1
4	5539	1
5	7746	1
6	1169	73

Hence $10001 = 7 * 37$ $\therefore$ Prime factors of 10001 are 73 and 37.

3. Pollard's Rho Algorithm

Algorithmic Perspective

- 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 (though the map generates random numbers, it has never been proved).
- Pollard's rho algorithm uses an iteration of the form: $x_{i+1} = f(x_i) \, mod \, n$, 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, $f(x) = x^2 + a$ is chosen (a $\neq 0$, -2 since they are fixed points for $f(x)$).

Into the Algorithm

The algorithm proceeds as follows :

Steps

STEP 1. : Compute $d = gcd\,(x - x', n)\,, \quad x' = f\,(x)$
STEP 2. : If $1 < d < n$, d is a proper divisor of n.
If d $= 1$, replace x by $f\,(x)$, and x' by $f\,(f\,(x'))$
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. $gcd\,(x_i - x_j, n) > 1$. This is reasoned out by the fact that if $gcd\,(x_i - n_j, n) > 1 \Rightarrow \; gcd\,(x_i - n_j, n)$ is a factor of n, which is either p or q.
- But whether we will get $gcd\,(x_i - n_j, n) > 1$ 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:

Analysis - Birthday Paradox and Pollard's Rho Algorithm

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. $x_i = x_j \, (mod \; p)$. Then $p \mid x_i - x_j$, and since $p \mid n$ also , we have $p \mid gcd\,(x_i - x_j, n)$.

Moreover since $x_i, x_j \in \mathbb{Z}_n$ and they are distinct
$\Rightarrow gcd\,(x_i - x_j, n) < n \Rightarrow \therefore gcd\,(x_i - x_j, n)$ 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 $gcd\,(x_i - x_j, n)$ for all pairs of $1 \leq i < j \leq k$, we find a non-trivial factor.

Analysis (Contd.)

- Now,
$$x_i \equiv x_j \, (mod \ p) \tag{1}$$
- The probability that equation (1) does not hold, i.e., $(x_1, x_2, \ldots, x_k) \, mod \ p$ are all different is:
$$\mathbb{P} = \left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right)\ldots\left(1 - \frac{k-1}{n}\right) \sim e^{\frac{-k^2}{2n}} \tag{2}$$
- 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 $\sqrt{n}$.

Analysis (Contd.)

Detecting Cycle

An optimisation that we can make here is to reduce the number of GCD computations by using Floyd's Cycle Detection Algorithm. Using this we will be able to deduce that there exists a pair $i < j < C_o\sqrt{p}$ (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

Pseudo Code

```
INITIALISE POLYNOMIAL MAP f
x  <- x_o
x' <- f(x) (mod n)
p  <- gcd(x - x', n)
while(p = 1)
{
    x  <- x_i
    x' <- x_2i
    x  <- f(x) (mod n)
    x' <- f(x')(mod n) { Executed Twice }
    p  <- gcd(x - x', n)
}
if(p = n)
    return "n is prime"
else
    return p
```

Time Complexity

Key Point

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 $\Theta\left(\sqrt{n}\right)$.

Example

- Let $f(x) = x^2 + 1$ be the sequence generator (our polynomial map).
- Factorise $n = 455459$; Let $x_o = 2$

Computation

$x = f(x)$	x' = f(f(x))% n	$p = \gcd(x - x', n)$
5	26	1
26	2871	1
677	179685	1
2871	155260	1
44380	416250	1
179685	43670	1
121634	164403	1
155260	247944	1
44567	68343	743

$\therefore n = 613 * 743$

4. Linear Sieve

Algorithmic Perspective

- 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 $O(n \log \log n)$.
- The Linear Sieve algorithm is similar to the classic Sieve of Eratosthenes but has linear complexity.
- Our goal is to calculate minimum prime factor $lp[i]$ for every number i in the segment $[2, n]$.
- We need to store the list of all the found prime numbers - let's call it $pr[]$.

Algorithm

- We'll initialize the values $lp[i]$ 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 : $lp[i] = 0$

This means that i is prime, i.e. we haven't found any smaller factors for it. Hence, we assign $lp[i] = i$ and add i to the end of the list $pr[]$.

Case 2 : $lp[i] \neq 0$

This means that i is composite, and its minimum prime factor is $lp[i]$.

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

Algorithm

Pseudo Code

```
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[]
```

Time & Space Complexity

Key Points

- Its time complexity is $O(n)$ which is slightly better than $O(n \log \log n)$.
- 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,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.

Example

- Let n = 10
- Initialise lp[2:10] to 0, and an empty list pr

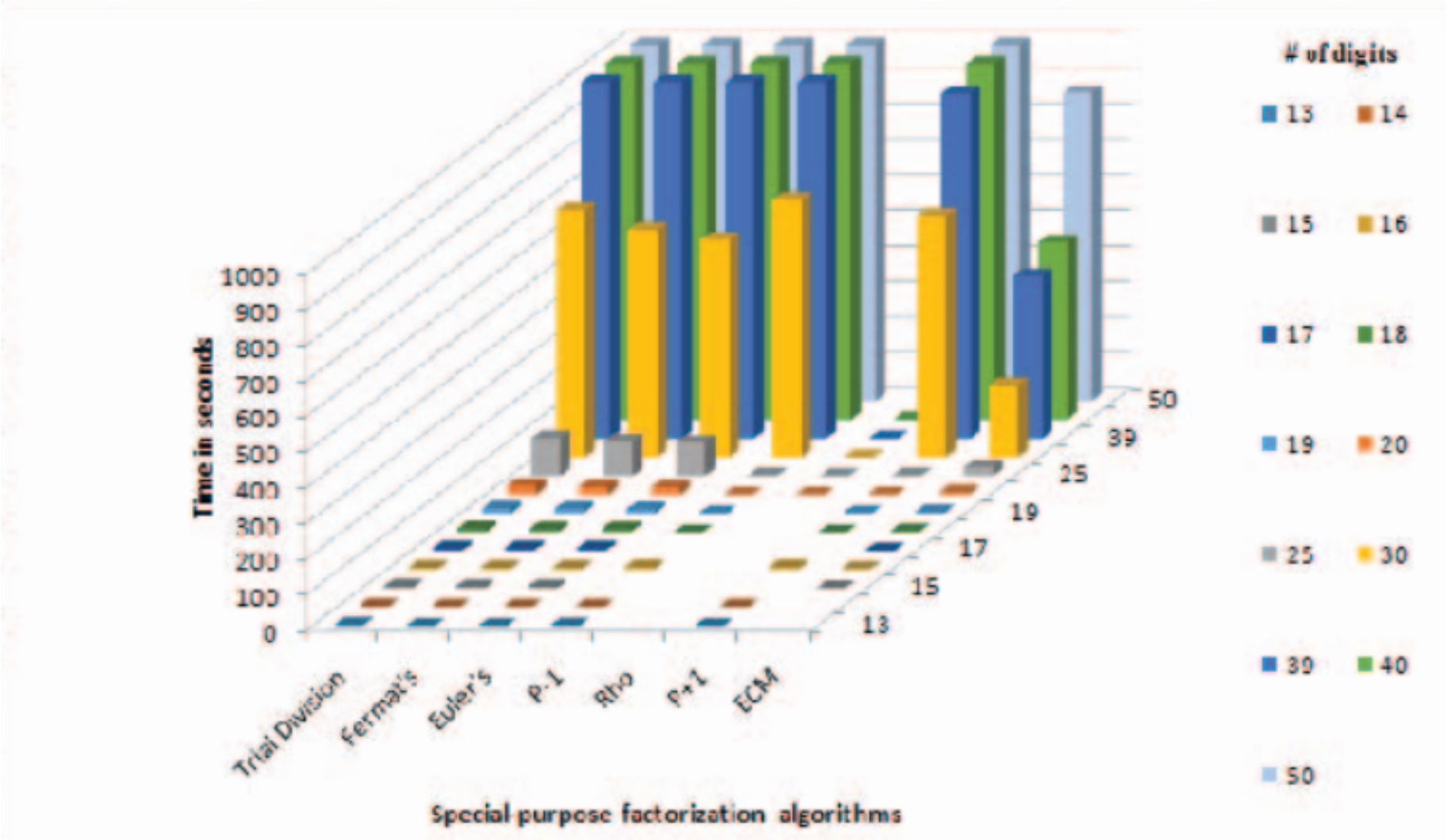
Computation

i	lp[](Outer Loop)	pr	j	lp[](Inner Loop)
2	lp[2] = 2	2	1	lp[4] = 2
3	lp[3] = 3	2, 3	1	lp[6] = 2
3	-	2, 3	2	lp[9] = 3
4	-	2, 3	1	lp[8] = 2
5	lp[5] = 5	2, 3, 5	-	-
6	-	2, 3, 5	-	-
7	lp[7] = 7	2, 3, 5, 7	-	-
8	-	2, 3, 5, 7	-	-
9	-	2, 3, 5, 7	-	-
10	-	2, 3, 5, 7	-	-

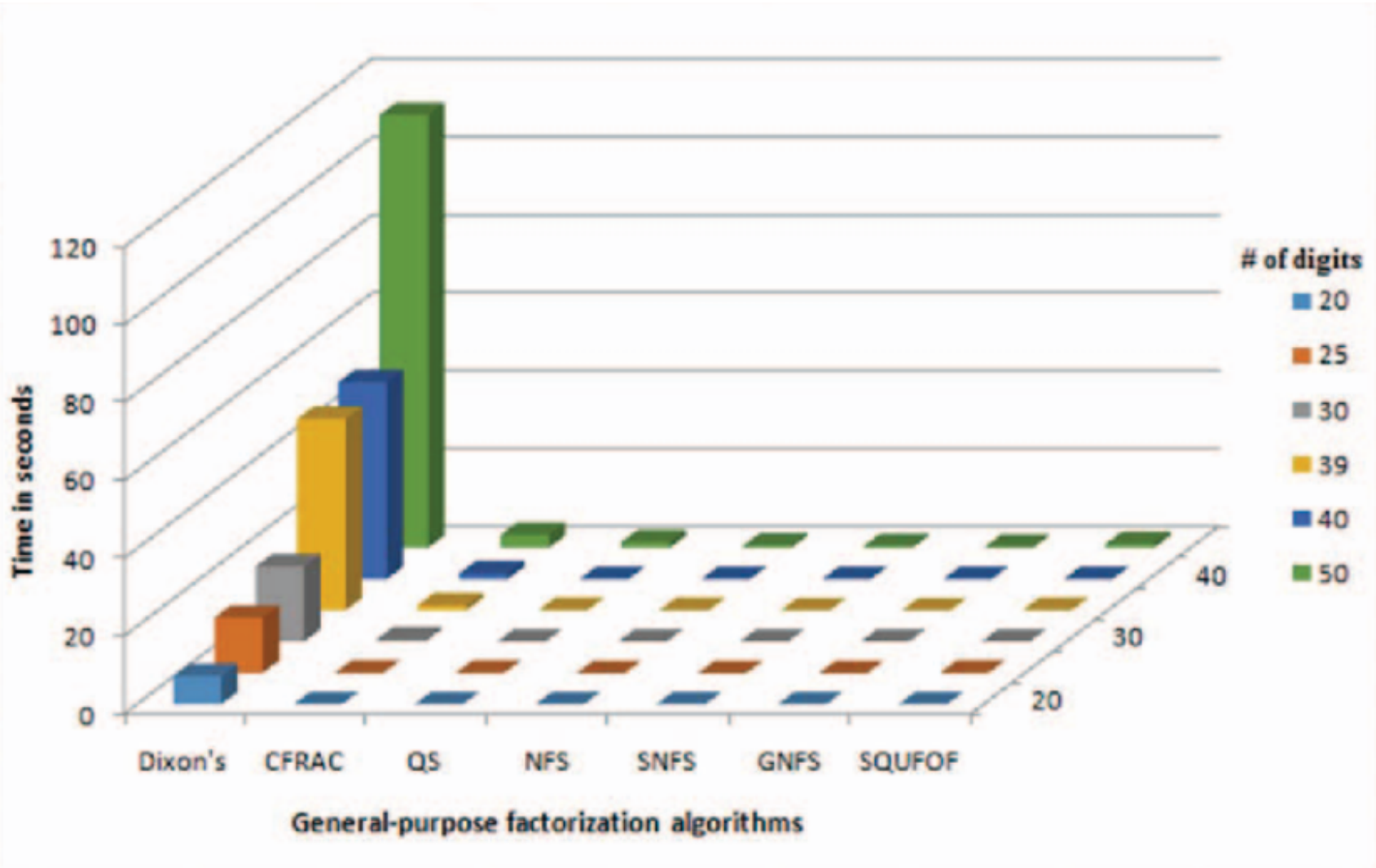
Conclusion

- 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.
- 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

Special Purpose Integer Factorisation Algorithms



General Purpose Integer Factorisation Algorithms



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*, (3*oth*June2009)
10. Cristina-Loredana Duta et al., *Framework for evaluation and*

Thank You