Assignment for the Postgraduate Course Cryptography: Factoring Large Numbers using Fermat's Method and it's variants.

Paschalis Aggelidis

Department of Mathematics

Aristotle University of Thessaloniki

August 4, 2023

Contents

Abstract

Fermat's well-known factorization algorithm is based on finding a representation of natural numbers n as the difference of squares. In 1974, R.Sherman Lehman modified Fermat's difference of squares method, to allow factorization of n in $O(n^{1/3})$ time. Moreover, a third method has been discovered which is also variant of Fermat's method, but quicker and less complex. We provide all of these in the following sections and some speed comparisons in the end.

***Keywords*—** factorization, speedrun, Fermat's Method, Lehman's Method, One Line Method

1 Introduction

Most modern methods of factoring are variants of Fermat's method of writing the number n to be factored as the difference of two squares, $n = (x - y)(x + y)$. In its simplest form, one starts with $y = \lfloor \sqrt{n} \rfloor$ and decrements y until $n - y^2$ is a square. Fermat's method is only practical if n has a factor very close to $\sqrt{n}$. The runtime complexity of Fermat's method is $O(n^{1/2+\epsilon})$, because there are up to $\sqrt{n}$ possible values of y. Lehman described a method for searching over a space of small fractions u/v that finds a factor of n in time $O(n^{1/3} + \epsilon)$. In the end, we will a describe a third method, which is a variant of Fermat's Algorithm and is somewhat similar to Lehman's Algorithm and compare it with the other two.

2 Fermat's Method of Factorization

2.1 Theoretical Backround of Fermat's Method

The first method we introduce is Fermat's and it is based in the following proposition [1].

Proposition 1. *Let n be an odd positive integer. Then there is a bijection between the factorizations of n in the form of $n = ab$, where a, b are integers with $a \geq b > 0$ and $t^2 - s^2$ of n, where s and t are integers ≥ 0. This bijection is given by the relations:*

$$t = \frac{a+b}{2}, \ s = \frac{a-b}{2}$$

and

$$a = t + s, \ b = t - s.$$

In order to find a factor of n we work as follows: We take $t = \lfloor \sqrt{n} \rfloor + 1, \ \lfloor \sqrt{n} \rfloor + 2, \ldots$ and we compute $t^2 - n$ until we find an integer s such that $t^2 - n = s^2$. Then, we will have $n = (t - s)(t + s)$.

If $n = ab$, then we need to examine all the positive integers until $(a + b)/2 - \lfloor \sqrt{n} \rfloor$. Thus, in case where integers a and b are really close, then $s = (a - b)/2$ is really small and $t = (a + b)/2$ is a bit bigger than $\lfloor \sqrt{n} \rfloor$. Then the method will give us the factorization of n after a small number of trials for t.

The following proposition gives an estimation for the number of steps required by Fermat's method, when the integer n is the product of two primes.

Proposition 2. *Let $n = pq$, where p and q primes and $p > q$, and A the number of steps required by Fermat's method to factorize n. Then,*

$$A = \left\lceil \frac{\left(\sqrt{p} - \sqrt{q}\right)^2}{2} \right\rceil < \frac{(p-q)^2}{8\lfloor \sqrt{n} \rfloor} + 1.$$

We conclude that if $n = pq$, where p and q are primes $p > q$, $p - q < \sqrt{8} n^{1/4}$, then Fermat's method factorizes n in one step.

If for every factorization $n = ab$ of n, the integers a and b are not close, then we will need a large number of trials for t. A speedup of the procedure would be to chose a small integer k and, by taking $t = \lfloor \sqrt{kn} \rfloor + 1, \ \lfloor \sqrt{kn} \rfloor + 2, \ldots,$ we calculate the values $t^2 - kn$ until we find an integer s such that, $t^2 - kn = s^2$. Then,

$$(t + s)(t - s) = kn$$

As the integers t and s are not close enough and k is small, we have $k < t - s < t + s < n$. Hence, there is a divisor δ of $t \pm s$ that does not divide k and as a result there is a prime divisor of δ that divides n. Thus, we have $1 < (t \pm s, n) < n$ and the integers $(t \pm s, n)$ are nontrivial factors of n.

2.2 The Algorithm of Fermat's Method

In this section we describe the algorithm based on Fermat's method of Factorization which is written and implemented in Python3.

Algorithm 1 Fermat's Difference of Squares Algorithm

procedure FERMAT(n)
  $s \leftarrow 0$
  while s is not square **do**
    $t \leftarrow \lfloor \sqrt{n} \rfloor + 1$
    $s \leftarrow t^2 - n$
    if s is square **then**
      return $t + s,\ t - s$
    end if
    $t \leftarrow t + 1$
  end while

The following code is an emplementation of the above algorithm in Python3.

[1]:
```
import math
import time

def fermat(n):
    t0 = math.isqrt(n)+1
    counter = 0
    t = t0+counter
    temp = math.isqrt((t*t)-n)
    while((temp*temp) != ((t*t)-n)):
        counter += 1
        t = t0+counter
        temp = math.isqrt((t*t)-n)
    s = temp
    p = t+s
    q = t-s
    return p
```

3 Lehman's Method

We will first describe Lehman's method of factorization [2]. We consider $x^2 - y^2 = 4kn$, $k = ab$ with $1 \leq k \leq r$. The idea is to divide the interval $[0, 1]$ into parts. Each part will correspond to a fraction a/b,

and these parts will fill the interval $[0, 1]$. This means that, for each r, we find a sequence S_r which includes a/b when $0 \le a \le b$, $b > 0$ and $ab \le r$.

Theorem 1. *Suppose that $n = ab$ where a and b are integers such that*

$$n^{1/3} < a \le n^{1/2} \le b < n^{2/3}.$$

Then there exists integers k, x and y such that

$$4kn = x^2 - y^2, \tag{1}$$

$$1 \le k \le n^{1/3} + 1, \tag{2}$$

$$(4kn)^{1/2} \le x \le (4kn)^{1/2} + \frac{n^{1/6}}{4k^{1/2}}, \tag{3}$$

$$0 \le y \le n^{1/3}. \tag{4}$$

Proof. We recall that Dirichlet's theorem asserts that if S is a real number, $S \ge 1$, and if θ is a real number, then there exists an integer s, $1 \le s \le S$, such that $\|\theta s\| \le 1/S$. Here $\|u\| = \min_{n \in \mathbb{Z}} |u - n|$ is the distance of u to the nearest integer. (This is the natural metric on the circle group $\mathbb{T} = \mathbb{R}/\mathbb{Z}$). Take $\theta = b/a$, $S = a/n^{1/3}$. Thus, there are integers r and s such that

$$\left| \frac{bs}{a} - r \right| \le \frac{n^{1/3}}{a}, \tag{5}$$

with $1 \le s \le a/n^{1/3}$. On multiplying by a, it follows that

$$|bs - ar| \le n^{1/3}. \tag{6}$$

From (5) we also see that $r > 0$, since $bs/a \ge 1$ and $n^{1/3} < 1$. Also from (5) we see that

$$r \le \frac{bs}{a} + \frac{n^{1/3}}{a}.$$

Since $s \le a/n^{1/3}$, it follows that

$$1 \le rs \le \frac{bs^2}{a} + \frac{n^{1/3}s}{a} \le n^{1/3} + 1.$$

We take $k = rs$, and note that (2) is satisfied. Set $x = ar + bs$, $y = |ar - bs|$. Then

$$x^2 - y^2 = (ar + bs)^2 - (ar - bs)^2 = 4arbs = 4kn.$$

Thus (1) is satisfied. By (6) we see that (4) is also satisfied. It remains to show that (3) holds. To this end we note that $x^2 = 4kn + y^2$. Hence

$$(4kn)^{1/2} \le x \le \left(4kn + y^2\right)^{1/2} \le \left(4kn + n^{2/3}\right)^{1/2} = (4kn)^{1/2} \left(1 + \frac{1}{4kn^{1/3}}\right)^{1/2}.$$

Thus, we (3) and the proof is complete. □

As a partial converse to Theorem 1, we prove

Theorem 2. *Suppose that $n \geq 8$ and that there exists k, x and y such that (1) − (4) hold. Then $(n, x+y)$ and $(n, x-y)$ are proper divisors of n.*

Proof. From (1) we see that $n \mid (x+y)(x-y)$. Thus, if $(n, x+y) = 1$, then $n \mid (x-y)$, and if $(n, x-y) = 1$, then $n \mid (x+y)$. Hence, it suffices to show that $0 < x - y \leq x + y < n$. From (2)-(4) we see that

$$y \leq n^{1/3} < (4n)^{1/2} \leq (4kn) \leq x.$$

Hence, $x - y > 0$. On the other hand, from (3) and (4) we see that

$$x + y \leq (4kn)^{1/2} + \frac{n^{1/6}}{4k^{1/2}} + n^{1/3}. \tag{7}$$

Let

$$f(u) = (4un)^{1/2} + \frac{n^{1/6}}{4k^{1/2}} + n^{1/3}. \tag{8}$$

It is easy to verify that f is increasing. Set $K = \left[n^{1/3} + 1\right]$. From (7) we deduce that

$$x + y \leq 2K^{1/2}n^{1/2} + \frac{n^{1/6}}{4K^{1/2}} + n^{1/3} \leq 2K^{1/2}n^{1/2} + n^{1/3} + \frac{1}{4}$$

since $K \geq n^{1/3}$. In the opposite direction we find that

$$K^{1/2} \leq \left(n^{1/3} + 1\right)^{1/2} = n^{1/6}\left(1 + \frac{1}{n^{1/3}}\right) \leq n^{1/6}\left(1 + \frac{1}{2n^{1/3}}\right) = n^{1/6} + \frac{1}{2n^{1/6}}.$$

Hence, $x + y \leq 2n^{2/3} + 2n^{1/3} + 1$. Let $g(u) = u^3 - 2u^2 - 2u - 1$. Since $g(u) = (u-3)(u^2+u+1) + 2$, it is clear that $g(u) > 0$ for $u \geq 3$. Hence $2n^{2/3} + 2n^{1/3} + 1 < n$, and the proof is complete. □

Suppose we know that h has no factor $\leq n^{1/3}$. Then, n is either prime, or the product of two primes. If the system (1)-(4) has no solution, then n is prime. If it has a solution, then the prime factors of n are the numbers $(n, x \pm y)$. To search for a solution, let $K = \left[n^{1/3} + 1\right]$. For each x in the interval (3) we test to see whether $x^2 - 4kn$ is a perfect square, say by the square-detecting algorithm. The number of tests is

$$\ll \sum_{k=1}^{K} \left(\frac{n^{1/6}}{k^{1/2}} + 1\right) \ll n^{1/6} + K^{1/2} + K \ll n^{1/3}. \tag{9}$$

3.1 Lehman's Algorithm

Lehman's Algorithm can be seen below.

Algorithm 2 Lehman's Factoring Algorithm

procedure LEHMAN(n)
  for $k \leftarrow 1, \ldots, \lceil n^{1/3} \rceil$ **do**
    for $a \leftarrow \lceil \sqrt{4kn} \rceil, \ldots, \lfloor \sqrt{4kn} + \frac{n^{1/6}}{4\sqrt{k}+1} \rfloor$ **do**
      $b \leftarrow a^2 - 4kn$
      if b is square **then**
        return GCD$(a + \sqrt{b}, n)$
      end if
    end for
  end for
  return 0

We now give the code of Lehman's Factoring method written in Python3.

[1]:
```
from itertools import accumulate, chain, cycle
from math import ceil, sqrt, floor, gcd, isqrt
from sympy.ntheory.primetest import is_square
import time

def td_factor(n, limit=10**6):
    'stop after first factor found'
    for f in accumulate(chain((2, 1, 2, 2), cycle((4, 2, 4, 2, 4, 6, 2, 6)))):
        if f*f > n or f > limit:
            return 0
        if n % f == 0:
            return f

def lehman(n):
    s3n = ceil(n ** (1/3))
    f = td_factor(n, limit=s3n)
    if f:
        return f
    for k in range(1, s3n + 1):
        sk = 2*sqrt(k*n)
        for a in range(ceil(sk), floor(sk + n**(1/6) / (4*sqrt(k)))+1):
            b = a*a - 4*k*n
            if is_square(b):
                return gcd(a + isqrt(b), n)
    return 0
```

4 One Line Factor Method

As we mentioned in the introduction, the algorithm is a variant of Lehman's algorithm if that n is given a mutliplier. However unlike, Lehman's algorithm, which applied Fermat's algorithm to nuv for various u/v, the only thing to be iterated in this algorithm is the multiplier itself [3]. The algorithm can be seen below.

Algorithm 3 One Line Algorithm

procedure OneLineFactor($n, iter$)
  for $i \leftarrow 1 \ldots iter$ **do**
    $s \leftarrow \lceil \sqrt{ni} \rceil$
    $m \leftarrow s^2 (\mathrm{mod} n)$
    if m is square **then**
      $t \leftarrow \sqrt{m}$
      return GCD$(n, s - t)$
    end if
  end for

An implementation of the One Line Algorithm can be seen in the following code written in Python3.

[1]:
```
import math
import time
from sympy.ntheory.primetest import is_square

def OneLineFactor(n):
    for i in range(1,math.floor(n**(1/3)+1)):
        s = math.ceil(math.sqrt(n*i))
        m = (s**2) % n
        if is_square(m):
            t = math.sqrt(m)
            return math.gcd(n, int(s-t))
```

In other words, we search for a solution to $t^2 = \left(\lceil \sqrt{ni} \rceil\right)^2 - ni$ by iterating i and looking for squares after reduction mod$n$.

4.1 Speedups of the Algorithm

A speedup of the algorithm can be obtained by multiplying n by a certain multiplier $M = \prod p_i^{n_i}$, for some small primes p_i, and applying the algorithm to Mn. We must ensure that n has been stripped of all its factors p_i by division before running algorithm. We avoid the factors p_i being returned by the algorithm by taking the GCD with n not Mn. Another speedup would be to reduce Mn at step 3, where we can simply subtract Mni from s^2. Larger multipliers mean that we have to look for a square after reduction mod$n$ the larger Mn or that we have to reduce mod$n$ instead of Mn, both of which are costly, thus it is not very practical to work with very large multipliers.

A further theoretical speedup would be to pick $C = \lceil \log \log n \rceil$ small primes q_i which are 1 mod 4. Then, search for prime numbers n_i which are squares modulo of all the q_i. By the law of quadratic reciprocity

the q_i are all squares modulo each of the n_i. Now replace step 4 of the algorithm with a step which checks whether m s a square modulo each of the n_i. If the n_i are small enough, this can be performed by table lookup.

This set will now pass any number which is of the form $\prod p_i^{m_i} t^2$ where $m_i \in \{0, 1\}$ and t is an nonnegative arbitrary number integer. We easily test whether m is really of this special form by removing factors of q_i by trial division and then testing whether the cofactor is a square. The algorithm is repeated until C such m are found. The quotient group $\mathbb{Z}/2\mathbb{Z}$, with the m_i as entries, can be used to find a product of such relation which yields a perfect square. This trick is a variant of Dixon's Method.

4.2 Analysis of the Algorithm

We will show that the algorithm has $O(n^{1/3+\epsilon})$ runtime. First of all, we assume that n has been trial factored up to $n^{1/3}$ which means $n^{1/3}$ iterations. This ensures that n has at most two prime factors, both of which are larger than $n^{1/3}$ and smaller than $n^{2/3}$. To simplify the matters that follow we will assume that the multiplier $M = 1$ and that n is not a perfect square.

Let $ni = u^2 + a$, where $0 < a < 2u + 1$. As n is not a perfect square and has no factors less than $n^{1/3}$ it is clear that $a > 0$. Then we have that $\lceil \sqrt{ni} \rceil^2 - ni = (u + 1)^2 - (u + a)^2 = 2u + 1 - a$. This is the value m in the algorithm. We have $0 < m \leq 2u < 2\sqrt{ni}$. We are searching for values for which m is a square. Then, there are approximately $\sqrt{2}(ni)^{1/4}$ squares less than $2\sqrt{ni}$. Thus the probability of finding a square at random is $1/\sqrt{2}(ni)^{1/4}$. Each iteration gives an independent chance of finding a square. So, if we complete $n^{1/3}$ iterations then i is bounded by $n^{1/3}$ so that each probability is at least $1/\sqrt{2}(ni)^{1/3}$ and that after $O(n^{1/3})$ iterations, in the limit, we are likely to factor n. We note that the largest factor our algorithm will find is $\lceil \sqrt{ni} \rceil + \sqrt{m}$; however, the first term is limited by $n^{2/3}$ and the second by $\sqrt{2}n^{1/3}$. In other words, the largest factor cannot be much bigger than $n^{2/3}$ if we do around $n^{1/3}$ iterations. Note that, this algorithm cannot find n as a factor, as it is too large, and it cannot return a factor of i, since the other factor must be a multiple of n.

4.3 Relationship to Lehman's factoring algorithm

Both the One Line Method and Fermat's Method can be improved through the use of multipliers. In Fermat's case where an integer n has three or more factors such as $n = pqr$, one of the factors can be produced at most $O(n^{1/3})$. In the remaining cases, if n has a factor then $n = pq$ for primes p, q. In this case, Fermat's algorithm won't find a factor quickly unless both of them are relatively close. However, if p/q is approximately equal to u/v for some small integers u, v then Fermat's algorithm will split $uvn = pv \times qu$ relatively quickly as it has factors close together.

Lehman's idea was to apply Fermat's algorithm to $4kn$ for multipliers k such that $0 < k \leq n^{1/3} + 1$. In other words, he wanted $4kn = x^2 - y^2$, and for any given k he showed that it was only necessary to check x such that $\sqrt{4kn} \leq x \leq \sqrt{4kn} + n^{1/6}/4k^{1/2}$. The main difference between Lehman's Algorithm and the One Line Algorithm is is that Lehman's Algorithm is way more complex than the One Liner, as both of them produce a factor of n in time $O(n^{1/3})$.

5 Numerical Results

In this section we test out both the Lehman's Algorithm and the One Line Algorithm and the results that were produced can be seen in the table above.

Number	Factor	Lehman's time is sec.	One Liner time in sec.
1123877887715932507	299155897	1.31E-1	1.12E-1
1129367102454866881	43655660929	9.42E-2	5.14E-2
29742315699406748437	372173423	1.23	5.9E-1
35249679931198483	59138501	6.34E-2	7.6E-2
208127655734009353	430470917	3.92E-1	1.54E-1
331432537700013787	114098219	2.83E-1	3.17E-1
3070282504055021789	2137748993	1.2	1.57E-1
3757550627260778911	16053127	1.38E-1	2.4E-2
24928816998094684879	71652460573	1.62	7.51E-1
10188337563435517819	143696355169	6.4E-1	3.66E-1

It is of no interest to test Fermat's algorithm since it is slow. Indeed, for the integer $n = 29742315699406748437$ would take too many trials since $(x-y,n)$ and $(x+y,n)$ are far far away from each other and Fermat's Algorithm is only practical when the two factors are close.

References

[1] Πουλάκης, "Υπολογιστική Θεωρία Αριθμών," 2015.

[2] R. S. Lehman, "Factoring large integers," *Mathematics of Computation*, vol. 28, no. 126, pp. 637–646, 1974.

[3] W. B. Hart, "A one line factoring algorithm," *Journal of the Australian Mathematical Society*, vol. 92, no. 1, pp. 61–69, 2012.