J. Aust. Math. Soc. **92** (2012), 61–69
doi:10.1017/S1446788712000146

A ONE LINE FACTORING ALGORITHM

WILLIAM B. HART

(Received 16 April 2011; accepted 1 February 2012)

Communicated by I. E. Shparlinski

In memory of my former PhD supervisor Alf van der Poorten

Abstract

We describe a variant of Fermat's factoring algorithm which is competitive with SQUFOF in practice but has heuristic run time complexity $O(n^{1/3})$ as a general factoring algorithm. We also describe a sparse class of integers for which the algorithm is particularly effective. We provide speed comparisons between an optimised implementation of the algorithm described and the tuned assortment of factoring algorithms in the Pari/GP computer algebra package.

2010 *Mathematics subject classification*: primary 11A51.
Keywords and phrases: integer factorisation, Lehman's algorithm, Fermat's algorithm.

1. Introduction

Most modern methods of factoring are variants of Fermat's method of writing the number n to be factored as a 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 run time complexity of Fermat's method is $O(n^{1/2+\epsilon})$, as there are up to $\sqrt{n}$ possible values of y.

If n has factors whose ratio is relatively close to the fraction u/v then applying Fermat's method to nuv will find the factors faster. Lehman [4] devised a method for searching over a space of small fractions u/v that finds a factor of n in time $O(n^{1/3+\epsilon})$.

More recently McKee [5] described a variant of Fermat's algorithm which can find a factor in expected time $O(n^{1/4+\epsilon})$. This method searches for solutions (x, y, z) to $z^2 = (x + \lceil\sqrt{n}\rceil y)^2 - ny^2$ with small x and y. The method achieves the stated run time complexity by observing that if m is an integer dividing z, then m^2 must divide the right-hand side of the equation. By computing a square root of $n \bmod m^2$ one can

Supported by EPSRC grants EP/G004870/1 and EP/D079543/1.

search for solutions in residue classes mod m^2. McKee gives timings which suggest that his algorithm is competitive with the factoring algorithms in Maple and Pari.

Most modern computer packages implement numerous algorithms for factoring. For numbers that fit into a single machine word, Shanks' SQUFOF (SQUare FOrms Factoring) algorithm (see [2, 7]) is popular as it has run time $O(n^{1/4})$ with a very small implied constant. As with McKee's algorithm, this is due to the fact that SQUFOF works with numbers of about half the bit size of n and that, in many cases, very few iterations are necessary to find a factor of n.

SQUFOF works by searching for *square forms* on the *principal cycle* of the binary quadratic forms of discriminant n or $4n$. A description of SQUFOF in terms of continued fractions and in terms of binary quadratic forms is given by Gower and Wagstaff in [2]. That paper also gives a set of heuristics for speeding up SQUFOF. The authors claim that SQUFOF is the 'clear champion factoring algorithm for numbers between 10^{10} and 10^{18}', at least on a 32-bit machine.

For larger numbers, subexponential algorithms such as the quadratic sieve and number field sieve are favoured, due to their lower asymptotic complexity.

In this paper we describe a variant of Fermat's algorithm which is somewhat similar in concept to Lehman's algorithm and compare it to the 'best of breed' factoring assortment in Pari/GP.

We call our algorithm the 'one line' factoring algorithm, as it can be implemented as a single (albeit long) line in some computer algebra systems, such as Pari/GP [6] or Sage [8]. (Our optimised implementation in C is obviously much more than one line.)

In a final section of the paper, we describe a sparse class of numbers which our algorithm is particularly efficient at factoring. In fact, numbers of many thousands of digits in this form may be factored easily by this algorithm.

2. Description of the factoring algorithm

We begin with a description of the algorithm. As mentioned in the introduction, the algorithm is a variant of Lehman's algorithm in that n is given a multiplier. 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.

```
OneLineFactor ($n$, $iter$)
    for $i \leftarrow 1 \dots iter$ do
        $s \leftarrow \lceil\sqrt{ni}\rceil$
        $m \leftarrow s^2 \pmod{n}$
        if IsSquare($m$) then
            $t \leftarrow \sqrt{m}$
            return GCD($n, s - t$)
        endif
    endfor
```

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

As advertised, the algorithm can be implemented in a single line of Pari code:

```
OLF(x)=;i=1;while(i<x,if(issquare(ceil(sqrt(i*x))^2%x),return \
(gcd(x,floor(ceil(sqrt(i*x))-sqrt((ceil(sqrt(i*x))^2)%x)))));i++)
```

As an example of its use, one may try

```
\p38 /* set precision to 38 digits */
n = nextprime(10^29+5342347)*nextprime(10^31+2232788)
OLF(n)
```

which will instantly produce the factors of n.

3. Practical and theoretical 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 prime factors p_i, and applying the algorithm to Mn. One must ensure that n has been stripped of all its factors of p_i by trial division before running the algorithm. One avoids the factors p_i being returned by the algorithm by taking the GCD with n not Mn.

Another immediate saving is made by noting that to reduce modulo Mn at step 3, one may simply subtract Mni from s^2.

In practice the multiplier $M = 480$ was observed to speed up the algorithm considerably as compared with a smaller multiplier or $M = 1$. Larger multipliers mean that we have to look for a square after reduction modulo the larger value Mn or that we have to reduce modulo n instead of Mn, both of which are costly, thus it is not practical to work with a very large multiplier.

A further theoretical speedup may be obtained as follows. Suppose that one wishes to factor a composite number n. Pick $C = \lceil \log \log n \rceil$ small primes q_i which are $1 \bmod 4$. Now perform a search for prime numbers n_i which are squares modulo all of 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 is a square modulo each of the n_i. If the n_i are small enough, this can be performed by table lookup.

This set of tests will now pass any number which is of the form $\prod q_i^{m_i} t^2$ where the m_i are in $\{0, 1\}$ and t is an arbitrary nonnegative integer. Numbers of this form are clearly more plentiful than squares.

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 'relations' m are found. Linear algebra over $\mathbb{Z}/2\mathbb{Z}$, with the m_i as entries, can then be used to find a product of such 'relations' which yields a perfect square.

This trick is essentially a variant of Dixon's method (see [1] for details).

It should be noted that whilst this trick yields more opportunities to factor n in theory, we did not observe a significant speedup in the range of practicality for this algorithm, that is, of general integers up to around $n = 2^{40}$. For numbers in this range, a square value m is found so quickly that it is rare that C relations are needed to factor n.

4. Heuristic analysis of the algorithm

We give a heuristic analysis of the algorithm showing that it has heuristic running time $O(n^{1/3+\epsilon})$.

First of all, we assume that n has been trial factored up to $n^{1/3}$. This takes $n^{1/3}$ iterations, which can clearly be done in the given run time. 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 analysis in what follows we assume that the fixed multiplier M is 1. We will also suppose n is not a perfect square. This can be checked before the algorithm begins.

Let us suppose that $ni = u^2 + a$ 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 $\lceil\sqrt{ni}\rceil^2 - ni = (u + 1)^2 - (u^2 + a) = 2u + 1 - a$. This is the value m in the algorithm.

Clearly we have $0 < m \leq 2u < 2\sqrt{in}$. We are searching for values for which m is a square. There are approximately $\sqrt{2}(ni)^{1/4}$ squares less than $2\sqrt{in}$. Thus the probability of hitting a square at random is $k(i) = 1/\sqrt{2}(ni)^{1/4}$.

We assume that each iteration gives an independent chance of finding a square.

If we complete $n^{1/3}$ iterations then i is bounded by $n^{1/3}$ so that each $k(i)$ is at least $1/\sqrt{2}(n)^{1/3}$. It is clear 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. However, as we have found all factors of n up to $n^{1/3}$ by trial factoring, then, assuming n is not prime, this condition is satisfied.

Note that the algorithm finds a *nontrivial* factor of n for a similar reason. It cannot find n as a factor, as it is too large, and it cannot return a factor of i, since the other factor in the difference of squares must then be a multiple of n.

5. Practical performance of the algorithm

In this section, we report on a relatively efficient implementation of the algorithm in the C language. This implementation is part of the FLINT [3] library.

We compare this implementation against the factor command in the Pari/GP package [6]. According to its documentation, Pari/GP has a highly developed assortment of factoring algorithms, including pure powers, trial division, Shanks' SQUFOF, Pollard–Brent Rho, Lenstra's ECM and a multiple polynomial quadratic sieve. It outputs Baillie-PSW pseudoprimes.

This implementation has been developed over many years and is considered highly optimised.

The constant multiplier M that we used in the implementation of our algorithm was 480. Arithmetic was performed in a single 64-bit machine word. To factor n of $3k$ bits,

TABLE 1. Comparison of One Line Factor and Pari.

Bits	One Line (μs)	Pari (μs)	Bits	One Line (μs)	Pari (μs)
4	0.10	0.26	22	4.48	2.60
5	0.11	0.26	23	5.59	3.21
6	0.13	0.36	24	5.77	4.16
7	0.16	0.36	25	7.36	5.68
8	0.18	0.43	26	9.11	7.14
9	0.20	0.43	27	9.91	9.86
10	0.22	0.44	28	12.7	13.6
11	0.26	0.48	29	15.5	23.0
12	0.32	0.55	30	18.2	33.2
13	0.43	0.57	31	22.8	44.4
14	0.47	0.64	32	28.9	58.5
15	0.68	0.71	33	32.8	71.5
16	0.82	0.78	34	41.2	88.5
17	1.05	0.94	35	50.2	95.7
18	1.51	1.10	36	61.6	116
19	1.83	1.30	37	74.4	133
20	2.25	1.59	38	99.6	162
21	3.90	2.06	39	115	171
22	4.48	2.60	40	145	194

we require about $4k + 9$ bits, as the multiplier i can be up to about $n^{1/3}$, and the constant multiplier is about 9 bits. Thus we expect to be able to factor integers up to around 41 bits without a significant number of failures. In fact, from 41 bits onward, a significant number of failures occurred, and below that point it was hard to find failures.

In Table 1, we give timings for our algorithm compared to that of the factor command in Pari/GP.

Because we are not interested in comparing the time for trial division, we first constructed an array of integers which had already been tested for small factors (up to $n^{1/3}$) using trial division. The two implementation then factored only numbers in this array. Thus neither implementation could crack the numbers using trial factoring up to $n^{1/3}$.

However, as the Pari/GP time would include significant overheads from trial factoring up to $n^{1/3}$, detecting perfect powers and from interpreter overhead, we ensured that our implementation still used trial factoring up to $n^{1/3}$ (even though no factors would be found at this step) and perfect power detection. We also timed the Pari/GP interpreter overhead separately and subtracted it from the Pari/GP timings for a fairer comparison.

We counted failures and found that our algorithm did not fail to factor integers in our test runs up to 40 bits. Thus our tables give timings up to that point. Beyond that point we would be merely comparing SQUFOF implementations, that being the fallback factoring routine in FLINT if the one line factoring algorithm fails. Furthermore, it becomes inefficient to check for factors up to $n^{1/3}$ from about this point on.

Timings were performed on a single core of a 2.4 GHz AMD K10 (Opteron) machine.

Clearly, in the range up to 26 bits, Pari wins by a small margin. However, factorisation in this range can be achieved purely with trial factoring up to $n^{1/2}$, which Pari does. We did not attempt to do this, as the speed of trial factoring is not relevant to our algorithm.

In the range where trial factoring alone is no longer appropriate, it is clear that our algorithm compares well with the factoring implementation in Pari. In this range, Pari is largely cracking composites with SQUFOF.

6. Relationship to Lehman's factoring algorithm

As mentioned in the introduction, Lehman noted that Fermat's algorithm can be improved through the use of multipliers.

In the case where n has three or more factors $n = pqr$, one of the factors is at most $n^{1/3}$. Thus it can be factored in time $O(n^{1/3})$.

In the remaining cases, if n has a factor then $n = pq$ for primes p, q. In this case, Fermat's algorithm will fail to find a factor quickly unless p and q are sufficiently 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 $\sqrt{4kn} \leq x \leq \sqrt{4kn} + n^{1/6}/4k^{1/2}$ (see [4] for details).

Lehman's algorithm has the advantage of guaranteeing a factor in time $O(n^{1/3})$. However, it has the distinct disadvantage of being more complex than our One Line Factor algorithm. In particular, the cost of computing the intervals which should be searched and the extra logic involved makes this a much slower algorithm in practice.

We developed a very careful implementation of Lehman's algorithm in C using the same fast square detection code and trial factoring routines that we developed for One Line Factor. Table 2 gives a timing comparison.

As can clearly be seen, the timings for One Line Factor are significantly lower, especially when the number being factored is large.

For the following observations we are indebted to the anonymous referee.

Note that as soon as $k > n^{1/3}/16$, Lehman's search interval contains just one integer.

One may therefore turn the loops in Lehman's algorithm inside-out, starting with one value of x from each interval and iterating the multiplier k. This gives an algorithm

Table 2. Comparison of One Line Factor and Lehman's algorithm.

Bits	One Line (µs)	Lehman (µs)
4	0.10	0.71
5	0.11	0.92
6	0.13	0.30
7	0.16	0.31
8	0.18	0.35
9	0.20	0.34
10	0.22	0.42
11	0.26	0.47
12	0.32	0.61
13	0.43	0.63
14	0.47	0.69
15	0.68	1.05
16	0.82	1.20
17	1.05	1.47
18	1.51	2.05
19	1.83	2.22
20	2.25	2.76
21	3.90	4.00
22	4.48	4.87
22	4.48	4.87
23	5.59	5.40
24	5.77	7.96
25	7.36	9.28
26	9.11	11.3
27	9.91	16.9
28	12.7	21.3
29	15.5	22.5
30	18.2	93.3
31	22.8	102
32	28.9	50.1
33	32.8	73.9
34	41.2	83.0
35	50.2	100
36	61.6	148
37	74.4	173
38	99.6	202
39	115	281
40	145	341

which is similar to One Line Factor. One Line Factor can thus be used to optimise the long tail of Lehman's algorithm.

In this way, One Line Factor can be turned into an algorithm which guarantees a factor. We implemented this trick and this algorithm gave the same times as the One Line Factor algorithm up to about 40 bits. This is due to the fact that One Line Factor rarely fails in this range so that it is almost always the only algorithm run.

7. Factorisations of a special form

The algorithm described in this paper was discovered whilst testing an implementation of the quadratic sieve. Various test composites were generated, and amongst them were ones of the form $n = \text{nextprime}(10^a) \times \text{nextprime}(10^b)$, for various values of a, b.

Interestingly, the algorithm in this paper factors numbers n of this form rather quickly. Once a, b are sufficiently large, Fermat's algorithm and its usual improvements are unable to factor numbers of this form in a reasonable time, unless additional information about the factors is known.

More generally, the one line factoring algorithm is able to deal with integers of the form $n = \text{nextprime}(c^a \pm d_1) \times \text{nextprime}(c^b \pm d_2)$ for $a > b$ and relatively close, and any small integers c, d_1, d_2. An example of such a factorisation is given in Section 2; however, numbers of many thousands of digits in this form can be factored.

As this class of integers is sparse, it is easy to concoct a contrived algorithm to factor such numbers. However, it is perhaps mildly interesting that this algorithm is completely general and still has this property.

We can see why our algorithm so easily cracks such composites if we recall the well-known fact that Fermat's algorithm will factor numbers of the form $n = \text{nextprime}(a) \times \text{nextprime}(a + j)$ for j less than approximately $a^{1/2}$ in a single iteration.

In the case of our algorithm, once the multiplier i reaches c^{a-b} we are essentially applying a single iteration of Fermat's algorithm to a number of this special form.

8. Summary

We have demonstrated a very simple-to-implement algorithm for factoring general integers in heuristic time $O(n^{1/3})$, with a very low implied constant.

We have demonstrated that this algorithm is very competitive for factoring numbers below about 42 binary bits (in combination with trial factoring) and can be used to quickly factor numbers in a certain sparse class, even when such numbers become very large.

Although the algorithm is not amenable to 'sieving' techniques, as per the quadratic and number field sieves, it may be useful in implementations of the large prime variants of such algorithms, as these require subordinate factorisations of small integers in their large prime phases. Often a highly optimised SQUFOF algorithm is used for this purpose, with which this algorithm is competitive.

Acknowledgements

Thanks to David Harvey for spotting some typos in an initial version of this paper. Thanks also to Thomas Boothby for supplying very fast perfect power detection code which was used in the final implementation. We are indebted to an anonymous referee for the suggestion at the end of Section 5.

References

[1] J. D. Dixon, 'Asymptotically fast factorization of integers', *Math. Comp.* **36** (1981), 255–260.
[2] J. Gower and S. Wagstaff Jr, 'Square form factorization', *Math. Comp.* **77** (2008), 551–588.
[3] W. Hart, 'FLINT: Fast Library for Number Theory', http://www.flintlib.org.
[4] R. Lehman, 'Factoring large integers', *Math. Comp.* **28** (1974), 637–646.
[5] J. McKee, 'Speeding Fermat's factoring method', *Math. Comp.* **68** (1999), 1729–1737.
[6] The Pari Group, 'Pari/GP', http://pari.math.u-bordeaux.fr/.
[7] D. Shanks, 'On Gauss and Composition II', in: *Number Theory and Applications (Banff, AB, 1988)*, NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci., 265 (ed. R. A. Mollin) (Kluwer, Dordrecht, 1989), pp. 179–204.

[8] W. Stein, ‘Can we create a viable free open source alternative to Magma, Maple, Mathematica and Matlab?’ *Proc. ISSAC 2008, Hagenberg, Austria, 20–23 July, 2008* (eds. J. R. Sendra and L. González-Vega) (ACM, New York, 2008), pp. 5–6. See also http://www.sagemath.org/.

WILLIAM B. HART, Zeeman Building, University of Warwick,
Coventry CV4 7AL, UK
e-mail: W.B.Hart@warwick.ac.uk