A ONE LINE FACTORING ALGORITHM

WILLIAM B. HART

Abstract. We describe a variant of Fermat's factoring algorithm which is competitive with SQUFOF in practice but has heuristic run time complexity $O(n^{\frac{1}{3}})$ 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 described algorithm, an optimised implementation of SQUFOF and the tuned assortment of factoring algorithms in the PARI computer algebra package.

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 $O(n^{\frac{1}{2}+\epsilon})$, as there are up to $\sqrt{n}$ possible values of y to check.

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 [**?**] devised a method for searching over a space of small fractions u/v that finds a factor of n in time $O(n^{\frac{1}{3}+\epsilon})$.

More recently McKee [2] described a variant of Fermat's algorithm which can find a factor in expected time $O(n^{\frac{1}{4}+\epsilon})$. 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 $n \pmod{m^2}$ one can search for solutions in residue classes $\pmod{m^2}$. 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 is popular as it has run time $O(n^{\frac{1}{4}})$ 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 the fact 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 [**?**]. 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 implementations of SQUFOF in PARI and our own implementation based on Gower and Wagstaff's heuristics. We also compare our implementation with a highly optimised version of the quadratic sieve which we have prepared.

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.

1. 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)
1   for i ← 1 ... iter do
2     s ← ⌈√(ni)⌉
3     m ← s^2 (mod n)
4     if issquare(m) then
5       t ← √m
6       return GCD(n, s − t)
7     endif
8   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.

We have called the algorithm OneLineFactor as it can be implemented in a single line of PARI code.

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.

2. Heuristic analysis of the algorithm

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

First of all we assume that n has been trial factored up to $n^{\frac{1}{3}}$. This takes $n^{\frac{1}{3}}$ 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 $n^{\frac{1}{3}}$ and smaller than $n^{\frac{2}{3}}$.

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 $n^{\frac{1}{3}}$ 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 $\sqrt{2}(ni)^{\frac{1}{4}}$ squares less than $2\sqrt{in}$. Thus the probability of hitting a square at random is $k(i) = 1/\sqrt{2}(ni)^{\frac{1}{4}}$.

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

If we complete $n^{\frac{1}{3}}$ iterations then i is bounded by $n^{\frac{1}{3}}$ so that each $k(i)$ is at least $1/\sqrt{2}(n)^{\frac{1}{3}}$. It is clear that after $n^{\frac{1}{3}}$ iterations, in the limit, there is a reasonable probability of hitting a square and thus factoring n.

We note that the largest factor our algorithm will find is $\lceil\sqrt{ni}\rceil + \sqrt{m}$, however the former is limited by $n^{\frac{2}{3}}$ and the latter by $\sqrt{2}n^{\frac{1}{3}}$. In other words the largest factor cannot be much bigger than $n^{\frac{2}{3}}$ if we do around $n^{\frac{1}{3}}$ iterations. However, as we have found all factors of n up to $n^{\frac{1}{3}}$ by trial factoring, then assuming n is not prime, this condition is satisfied.

Note that the algorithm finds a non-trivial 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.

3. Acknowledgements

Thanks to David Harvey for spotting some typos in an initial version of this paper.

References

[1]
[2]
[3]

E-mail address: W.B.Hart@warwick.ac.uk