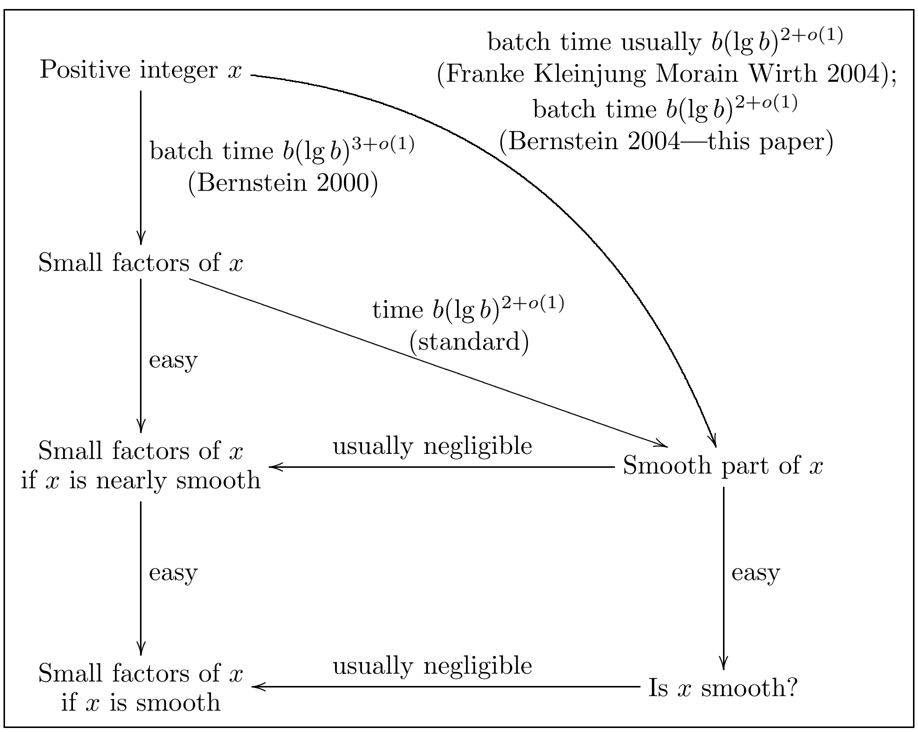
Draft. Aimed at Math. Comp. I'm rewriting [8] in light of this.

HOW TO FIND SMOOTH PARTS OF INTEGERS

DANIEL J. BERNSTEIN

Abstract. Let P be a finite set of primes, and let S be a finite sequence of positive integers. This paper presents an algorithm to find the largest P-smooth divisor of each integer in S. The algorithm takes time $b(\lg b)^{2+o(1)}$, where b is the total number of bits in P and S. A previous algorithm by the author takes time $b(\lg b)^{3+o(1)}$ to find all the factors from P of each integer in S; a variant by Franke, Kleinjung, Morain, and Wirth *usually* takes time $b(\lg b)^{2+o(1)}$ to find the largest P-smooth divisor of each integer in S; the algorithm in this paper *always* takes time $b(\lg b)^{2+o(1)}$ to find the largest P-smooth divisor of each integer in S.

1. Introduction



Date: 2004.05.10. Permanent ID of this document: `201a045d5bb24f43f0bd0d97fcf5355a`.
2000 *Mathematics Subject Classification.* Primary 11Y16.
The author was supported by the National Science Foundation under grant DMS–0140542.

This paper presents an algorithm that, given a finite set P of primes and a finite sequence S of positive integers, identifies the P-smooth elements of S. Here a positive integer is **P-smooth** if it is a product of powers of elements of P. The algorithm takes time $b(\lg b)^{2+o(1)}$, where b is the total number of bits in P and S.

The algorithm actually obtains more information: namely, the P-smooth part of each element of S. Here the **P-smooth part** of a positive integer is the largest P-smooth divisor of that integer; for example, the $\{2,3,5\}$-smooth part of $2^3 5^1 7^1 23^1$ is $2^3 5^1$.

Section 2 presents the algorithm. Section 3 presents various improvements in the $o(1)$. Another section will present details of, and concrete speed reports for, an improved algorithm.

Genealogy. My previous algorithm in [8] produces more information, namely all the factors from P of each element of S, but takes time $b(\lg b)^{3+o(1)}$.

Franke, Kleinjung, Morain, and Wirth in [21] introduced an algorithm variant that *usually* takes time $b(\lg b)^{2+o(1)}$ to find the largest P-smooth divisor of each element of S. The algorithm takes much more time for nasty inputs.

The algorithm in this paper is a slight further variant that *always* takes time $b(\lg b)^{2+o(1)}$; see Section 2. In typical applications, the algorithm in this paper is slightly faster than the algorithm of Franke et al.; see Section 3.

Competition. There are several previous algorithms that find the P-smooth part of each element of S separately, in the important special case that P is the set of prime numbers below some limit:

- Trial division takes time at most $b^{2+o(1)}$.
- Pollard's fast-factorial method in [29] takes time at most $b^{1.5+o(1)}$.
- Conjectured to work: Pollard's rho method in [30] takes time at most $b^{1.5+o(1)}$, with a smaller $o(1)$ than in [29]. See [14] and [15] for improvements, and [3] for some progress towards proving the conjecture.
- Conjectured to work: Lenstra's smooth-sized-elliptic-curve method in [23], improving upon Pollard's smooth-$(p-1)$ method in [29] and Williams's smooth-$(p+1)$ method in [37], takes time $b^{1+o(1)}$—more precisely, time at most $b \exp\sqrt{(2+o(1))\log b \log\log b}$. For further discussion see [16], [27], [22], [17], [28], [1], [35], [31], [13], and [18]. The variants in [2] and [24] appear to be slower, although the variant in [24] has the virtue of being *proven* to work with negligible error probability.

None of these methods can be reasonably conjectured—never mind proven—to work in time $b(\lg b)^{O(1)}$ for typical input distributions. Furthermore, in practice, none of these methods are competitive with the algorithm in this paper.

Applications. Given a finite set P of primes and a finite sequence S of positive integers, one can identify and factor the P-smooth elements of S as follows:

- Use the algorithm in this paper to compute the P-smooth part of each element of S. This takes time $b(\lg b)^{2+o(1)}$.
- List the elements of S that equal their P-smooth parts. These are the P-smooth elements of S.
- Use the algorithm in [8] to factor the P-smooth elements of S. This takes time at most $b(\lg b)^{2+o(1)}$ if the smooth elements, together with P, occupy at most $b/(\lg b)^{1+o(1)}$ bits. In typical applications, S is substantially larger

than P, and only a tiny fraction of the elements of S are smooth, so this step takes negligible time.

This type of computation—identifying and factoring the P-smooth elements of a sequence—is a bottleneck in the Lehmer-Powers-Brillhart-Morrison continued-fraction method of factoring integers, and in many newer algorithms for factoring integers, computing discrete logarithms, computing regulators, etc. See, e.g., [32].

In the same way, one can identify and factor the elements of S that are **nearly smooth**: smooth numbers times apparent prime numbers. This is a bottleneck in proving primality with elliptic curves. Franke, Kleinjung, Morain, and Wirth stated their algorithm in this context. See [21].

Sieving. In many applications, S is the sequence of values $f(0), f(1), f(2), \ldots$ of a low-degree polynomial f on consecutive inputs $0, 1, 2, \ldots$. The goal, typically, is to find a specified number of smooth values of f as quickly as possible.

For each prime p, the set of i such that p divides $f(i)$ is a union of a small number of arithmetic progressions. **Sieving** zooms through those arithmetic progressions to compute the factors from P of all $f(i)$'s simultaneously.

Sieving can be profitably combined with non-sieving algorithms, such as the algorithm in this paper, if P is not very small. The combination is explained and analyzed in my companion paper [11]. The bottom line is that each order-of-magnitude speedup in non-sieving algorithms produces a somewhat smaller speedup in the combined algorithm.

A note on models of computation. I am, of course, measuring algorithm time in the traditional way, as the number of steps on a conventional von Neumann computer, i.e., on a processor with fast access to a large bank of memory.

This is not the most cost-effective architecture for large computers. There is a huge literature showing that mesh architectures achieve better price-performance exponents than von Neumann architectures for a wide variety of problems. I pointed out in 2001 that smoothness detection was one of those problems; see [4], [5], and [6]. In the long run, the analysis of algorithm time on von Neumann architectures will be far less important than the analysis of algorithm cost on mesh architectures. For the moment, however, conventional von Neumann computers are sufficiently popular to justify continued analysis of their capabilities.

2. The algorithm

Algorithm 2.1. Given prime numbers $p_1, \ldots, p_m$ and positive integers $x_1, \ldots, x_n$, to print the $\{p_1, \ldots, p_m\}$-smooth part of each x_k:

1. Compute $z \leftarrow p_1 \cdots p_m$ using a product tree.
2. Compute $z \bmod x_1, \ldots, z \bmod x_n$ using a remainder tree.
3. For each $k \in \{1, \ldots, n\}$: Compute $y_k \leftarrow (z \bmod x_k)^{2^e} \bmod x_k$ by repeated squaring, where e is the smallest nonnegative integer such that $2^{2^e} \geq x_k$.
4. For each $k \in \{1, \ldots, n\}$: Print $\gcd\{x_k, y_k\}$.

Theorem 2.2. *Algorithm 2.1 prints the $\{p_1, \ldots, p_m\}$-smooth part of each x_k.*

One can generalize Algorithm 2.1 to arbitrary positive integers $p_1, \ldots, p_m$. Then the kth output is the largest divisor of x_k that is a product of powers of primes dividing $p_1 \cdots p_m$.

Proof. The kth output is $\gcd\{x_k, y_k\} = \gcd\big\{x_k, z^{2^e}\big\}$ where e is the smallest nonnegative integer such that $2^{2^e} \ge x_k$.

If q is a prime number outside $\{p_1, \ldots, p_m\}$ then $\operatorname{ord}_q z = 0$ so $\operatorname{ord}_q \gcd\{x_k, z^{2^e}\} = 0$. Thus the kth output is a product of powers of $\{p_1, \ldots, p_m\}$.

If $q \in \{p_1, \ldots, p_m\}$ then $\operatorname{ord}_q z^{2^e} \ge 2^e \ge \operatorname{ord}_q x_k$. (If $\operatorname{ord}_q x_k$ were larger than 2^e then x_k would be larger than $q^{2^e} \ge 2^{2^e}$.) Hence $\operatorname{ord}_q \gcd\big\{x_k, z^{2^e}\big\} = \operatorname{ord}_q x_k$. Thus the kth output divides x_k, and the quotient is not divisible by any of $p_1, \ldots, p_m$. □

Theorem 2.3. *Algorithm 2.1 takes time $O(b(\lg b)^2 \lg\lg b)$ where b is the number of input bits.*

Proof. Step 1 takes time $O(b(\lg b)^2 \lg\lg b)$. See [9, Section 12].

Step 2 takes time $O(b(\lg b)^2 \lg\lg b)$ since $z, x_1, x_2, \ldots, x_n$ together have $O(b)$ bits. See [9, Section 18].

Write b_k for the number of bits in x_k. Then the computation of y_k in Step 3 takes time $O(b_k(\lg b)^2 \lg\lg b)$ since $e \in O(\lg b)$. The total over all k is $O(b(\lg b)^2 \lg\lg b)$.

The computation of $\gcd\{x_k, y_k\}$ in Step 4 takes time $O(b_k(\lg b)^2 \lg\lg b)$. See [9, Section 22]. The total over all k is $O(b(\lg b)^2 \lg\lg b)$. □

This proof suggests that the speed of each step is critical. However, Step 3 and Step 4 are not bottlenecks in the typical case that each x_k has far fewer than b bits.

History. My batch factorization algorithm in [8]

- computes $x_1 \cdots x_n$;
- computes $(x_1 \cdots x_n) \bmod p_1, \ldots, (x_1 \cdots x_n) \bmod p_m$;
- discards the primes that don't divide $x_1 \cdots x_n$;
- chops the sequence $x_1, \ldots, x_m$ in half; and
- handles each half recursively.

Franke, Kleinjung, Morain, and Wirth in [21] introduced a "simplified version" of my algorithm in the context of batch smoothness detection for proving primality with elliptic curves. In fact, their algorithm has a different structure: it begins by computing $(p_1 \cdots p_m) \bmod x_1, \ldots, (p_1 \cdots p_m) \bmod x_n$, and then handles each x_k independently. Algorithm 2.1 follows this structure.

(Here is another potentially useful structure, following the philosophy that the relevant primes should be discovered, as in [7], rather than specified in advance: compute $((x_1 \cdots x_n)/x_1) \bmod x_1$, $((x_1 \cdots x_n)/x_2) \bmod x_2$, etc. One fast way to do this, suggested by Borodin and Moenck, is to first compute $(x_1 \cdots x_n) \bmod x_1^2$, $(x_1 \cdots x_n) \bmod x_2^2$, etc.; see [9, Section 23] for further discussion.)

To handle x_k, Franke et al. repeatedly replace x_k by $x_k/\gcd\{p_1 \cdots p_m, x_k\}$ until $\gcd\{p_1 \cdots p_m, x_k\} = 1$; the ratio between the original x_k and the final x_k is the smooth part of the original x_k. The general problem of computing $\gcd\{x, z^\infty\}$ has appeared in several contexts other than batch smoothness detection; the strategy used by Franke et al. is the same as the strategy used in, e.g., [26]. The problem with this strategy is that the number of iterations can be very large.

One way to limit the number of iterations is to square z after each iteration, as in, e.g., [7, Section 11] and [34, page 797]. Algorithm 2.1 instead does several squarings so that a single gcd suffices, as in [33]. See Section 3 for comments on combined strategies.

3. Speedups

This section describes several ways to save time in Algorithm 2.1. These speedups are not visible at the level of detail of $b(\lg b)^{2+o(1)}$ but are nevertheless valuable in practice.

This section is only an outline for the moment; I'll add more comments later.

Removing redundancy in product trees. Robert Kramer has recently introduced a technique, which I call "FFT doubling," that saves time in the computation of product trees. The speedup factor is $1.5+o(1)$ for a large balanced product tree.

Balancing trees. Here I'll comment on Strassen's speedup when the entropy of the input size distribution is unusually small.

Removing redundancy in division. One can use FFT caching, FFT addition, etc. to save time inside (and outside) division. See [10].

Allowing a wider remainder range. Instead of forcing remainders to be in the range $\{0, 1, \ldots, x_k - 1\}$ (or a similar range balanced around 0), one can allow remainders to be a few bits larger than x_k. This saves a surprising amount of time in division.

Removing redundancy in remainder trees. The remainder-tree computation involves, for example, computing approximate reciprocals of x_1, x_2, and x_1x_2. For x_1x_2 one should start Newton's method at the product of approximate reciprocals of x_1 and x_2, rather than at 1.

Reducing the exponent. Christine Swart has pointed out that the exponent e in Step 3 can be reduced: for example, instead of using $z = 2\cdot 3\cdot 5\cdots$ with $2^{2^e} \geq x_k$, one can use $z = 16 \cdot 27 \cdot 25 \cdots$ with $16^{2^e} \geq x_k$.

Skipping the gcd. In many applications, one simply wants to know whether x_k is smooth. Step 4 can then be simplified: one has $\gcd\{x_k, y_k\} = x_k$ if and only if $y_k = 0$.

Balancing gcd and powering. One can compute a moderate power, then a gcd, then a moderate power, then a gcd, and so on. I'll comment here on the speed ratio in practice between the optimum combination and the all-gcd approach of Franke et al.

Eliminating tiny primes. A very small amount of trial division is helpful. In particular, one can save time by removing all factors of 2 from each x_k.

The 2-adic variant. 2-adic division is slightly faster and simpler than real division.

References

[1] A. O. L. Atkin, Francois Morain, *Finding suitable curves for the elliptic curve method of factorization*, Mathematics of Computation **60** (1993), 399–405. ISSN 0025–5718. MR 93k:11115.

[2] Eric Bach, Jeffrey Shallit, *Factoring with cyclotomic polynomials*, Mathematics of Computation **52** (1989), 201–219. ISSN 0025–5718. MR 89k:11127.

[3] Eric Bach, *Toward a theory of Pollard's rho method*, Information and Computation **90** (1991), 139–155. ISSN 0890–5401. MR 92a:11151.

[4] Daniel J. Bernstein, *The NSA sieving circuit* (2001). URL: http://cr.yp.to/talks.html#2001.05.07.

[5] Daniel J. Bernstein, *An introduction to Schimmler sorting* (2001). URL: http://cr.yp.to/talks.html#2001.05.14.

[6] Daniel J. Bernstein, *Circuits for integer factorization: a proposal* (2001). URL: http://cr.yp.to/papers.html.

[7] Daniel J. Bernstein, *Factoring into coprimes in essentially linear time*, to appear, Journal of Algorithms. ISSN 0196–6774. URL: http://cr.yp.to/papers.html. ID f32943f0bb67a9317d4021513f9eee5a.

[8] Daniel J. Bernstein, *How to find small factors of integers*, accepted to Mathematics of Computation; now being revamped. URL: http://cr.yp.to/papers.html.

[9] Daniel J. Bernstein, *Fast multiplication and its applications*, to appear in Buhler-Stevenhagen *Algorithmic number theory* book. URL: http://cr.yp.to/papers.html#multapps.

[10] Daniel J. Bernstein, *Removing redundancy in high-precision Newton iteration*, draft. URL: http://cr.yp.to/papers.html#fastnewton. ID def7f1e35fb654671c6f767b16b93d50.

[11] Daniel J. Bernstein, *Sieving in cache*, draft. URL: http://cr.yp.to/papers.html#cachesieving.

[12] Wieb Bosma, Alf J. van der Poorten (editors), *Computational algebra and number theory: CANT2*, Kluwer Academic Publishers, Dordrecht, 1995. ISBN 0–7923–3501–5. MR 96c:00019.

[13] Wieb Bosma, Arjen K. Lenstra, *An implementation of the elliptic curve integer factorization method*, in [12] (1995), 119–136. MR 96d:11134.

[14] Richard P. Brent, *An improved Monte Carlo factorization algorithm*, BIT **20** (1980), 176–184. ISSN 0006–3835. MR 82a:10017.

[15] Richard P. Brent, John M. Pollard, *Factorization of the eighth Fermat number*, Mathematics of Computation **36** (1981), 627–630. ISSN 0025–5718. MR 83h:10014.

[16] Richard P. Brent, *Some integer factorization algorithms using elliptic curves*, Australian Computer Science Communications **8** (1986), 149–163. ISSN 0157–3055.

[17] Richard P. Brent, *Parallel algorithms for integer factorisation*, in [25] (1990), 26–37. MR 91h:11148.

[18] Richard P. Brent, *Factorization of the tenth Fermat number*, Mathematics of Computation **68** (1999), 429–451. ISSN 0025–5718. MR 99e:11154.

[19] Jacques Calmet (editor), *Algebraic algorithms and error-correcting codes 3*, Lecture Notes in Computer Science, 229, Springer-Verlag, Berlin, 1986. ISBN 0387167765.

[20] Srishti D. Chatterji (editor), *Proceedings of the International Congress of Mathematicians*, Birkhauser Verlag, Basel, 1995. ISBN 3–7643–5153–5. MR 97c:00049.

[21] Jens Franke, T. Kleinjung, François Morain, T. Wirth, *Proving the primality of very large numbers with fastECPP*. URL: ftp://lix.polytechnique.fr/pub/submissions/morain/Preprints/large.ps.gz.

[22] James L. Hafner, Kevin S. McCurley, *On the distribution of running times of certain integer factoring algorithms*, Journal of Algorithms **10** (1989), 531–556. ISSN 0196–6774. MR 91g:11157.

[23] Hendrik W. Lenstra, Jr., *Factoring integers with elliptic curves*, Annals of Mathematics **126** (1987), 649–673. ISSN 0003–486X. MR 89g:11125. URL: http://links.jstor.org/sici?sici=0003-486X(198711)2:126:3<649:FIWEC>2.0.CO;2-V.

[24] Hendrik W. Lenstra, Jr., Jonathan Pila, Carl Pomerance, *A hyperelliptic smoothness test, I*, Philosophical Transactions of the Royal Society of London Series A **345** (1993), 397–408. ISSN 0962–8428. MR 94m:11107. URL: http://links.jstor.org/sici?sici=0962-8428(19931115)345:1676<397:AHSTI>2.0.CO;2-P.

[25] John H. Loxton (editor), *Number theory and cryptography*, London Mathematical Society Lecture Note Series, 154, Cambridge University Press, Cambridge, 1990. ISBN 0–521–39877–0. MR 90m:11003.

[26] Heinz Lüneburg, *On a little but useful algorithm*, in [19] (1986), 296–301. URL: `http://cr.yp.to/bib/entries.html#1986/lueneburg`.

[27] Peter L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Mathematics of Computation **48** (1987), 243–264. ISSN 0025–5718. MR 88e:11130.

[28] Peter L. Montgomery, *An FFT extension of the ellpitic curve method of factorization*, Ph.D. thesis, University of California at Los Angeles, 1992.

[29] John M. Pollard, *Theorems on factorization and primality testing*, Proceedings of the Cambridge Philosophical Society **76** (1974), 521–528. ISSN 0305–0041. MR 50:6992. URL: `http://cr.yp.to/bib/entries.html#1974/pollard`.

[30] John M. Pollard, *A Monte Carlo method for factorization*, BIT **15** (1975), 331–334. ISSN 0006–3835. MR 52:13611. URL: `http://cr.yp.to/bib/entries.html#1975/pollard`.

[31] Carl Pomerance, Jonathan Sorenson, *Counting the integers factorable via cyclotomic methods*, Journal of Algorithms **19** (1995), 250–265. ISSN 0196–6774. MR 96e:11163. URL: `http://cr.yp.to/bib/entries.html#1995/pomerance-cyclo`.

[32] Carl Pomerance, *The role of smooth numbers in number-theoretic algorithms*, in [20] (1995), 411–422. MR 97m:11156.

[33] Arnold Schönhage, *Fast reduction and composition of binary quadratic forms*, in [36] (1991), 128–133.

[34] Joseph H. Silverman, *Computing canonical heights with little (or no) factorization*, Mathematics of Computation **66** (1997), 787–805. ISSN 0025–5718. MR 97f:11040. URL: `http://www.ams.org/journal-getitem?pii=S0025571897008120`.

[35] Robert D. Silverman, Samuel S. Wagstaff, Jr., *A practical analysis of the elliptic curve factoring algorithm*, Mathematics of Computation **61** (1993), 445–462. ISSN 0025–5718. MR 93k:11117.

[36] Stephen M. Watt (editor), *Proceedings of the 1991 international symposium on symbolic and algebraic computation, July 15–17, 1991, Bonn, West Germany*, Association for Computing Machinery, New York, 1991. ISBN 0–89791–437–6.

[37] Hugh C. Williams, *A $p + 1$ method of factoring*, Mathematics of Computation **39** (1982), 225–234. ISSN 0025–5718. MR 83h:10016.

Department of Mathematics, Statistics, and Computer Science (M/C 249), The University of Illinois at Chicago, Chicago, IL 60607–7045, USA

E-mail address: `djb@cr.yp.to`