MATHEMATICS OF COMPUTATION
VOLUME 41, NUMBER 163
JULY 1983, PAGES 287–294

Factoring Large Numbers With a Quadratic Sieve

By Joseph L. Gerver

Abstract. The quadratic sieve algorithm was used to factor a 47-digit number into primes. A comparison with Wagstaff's results using the continued fraction early abort algorithm suggests that QS should be faster than CFEA when the number being factored exceeds 60 digits (plus or minus ten or more digits, depending on details of the hardware and software).

I. Introduction. Pomerance [7] has shown that, if one makes certain reasonable but unproved assumptions, his quadratic sieve algorithm (QS) is asymptotically faster than any other known general algorithm for factoring large numbers, with the possible exception of a very recent algorithm of Schnorr [8]. In particular, for sufficiently large numbers, the quadratic sieve algorithm is faster than the continued algorithm (CF) of Brillhart and Morrison [6], even if the latter includes the early abort modification (CFEA).

I have recently used the quadratic sieve algorithm to factor the 47-digit number 176749718190056652686820090382275793007611620l, a factor of $3^{225} - 1$ which was known [1] to be composite but had never been factored, into three prime factors: 119634969443826601, 286870274711101, and 515009259868501. Wagstaff [9] has recently used both CF and CFEA to factor several 47-digit numbers. A comparison of the execution times for these programs indicates that QS becomes faster than CF when the number to be factored exceeds 40 digits, and faster than CFEA when the number exceeds 60 digits. However, the ratio of the running time of QS to each of the other algorithms is quite insensitive to the size of the factored number. Thus the break-even points of 40 and 60 digits could easily be off by ± 10 digits, depending on details of the hardware and software.

II. Description of the Algorithms. All modern general factoring algorithms (i.e. algorithms which do not require the factored number to be of any special form) are based on Fermat's observation that a composite number n can be factored if one can find two squares $x^2 \equiv y^2 \pmod{n}$, such that $x \not\equiv \pm y \pmod{n}$. For then $n \mid (x^2 - y^2) = (x + y)(x - y)$ but n does not divide $x + y$ or $x - y$; hence the greatest common divisors $(n, x + y)$ and $(n, x - y)$ are proper factors of n.

To find two such squares, one starts with a factor base of k primes, $p_1, \ldots, p_k$, and then searches for sequences of integers $a_1, \ldots, a_{k+1}$ and $b_1, \ldots, b_{k+1}$ such that, for each i, $a_i^2 \equiv b_i \pmod{n}$ and b_i can be completely factored over the factor base; i.e.

$$b_i = \prod_{j=1}^{k} p_j^{c_{ij}}.$$

Received June 24, 1982.
1980 *Mathematics Subject Classification.* Primary 10A25; Secondary 10–04, 68C25.

0025-5718/83/0000-1432/$04.75

For each i, let $\vec{v}_i = (v_{i1}, \ldots, v_{ik})$ be the vector over the integers mod 2 such that, for each j, $v_{ij} \equiv c_{ij} \pmod 2$. There must exist a linear dependency among the vectors $\vec{v}_1, \ldots, \vec{v}_{k+1}$, and hence a set S such that $\Sigma_{i \in S} \vec{v}_i = 0$. Therefore $\Pi_{i \in S} b_i$ is a perfect square, y^2, which is congruent (mod n) to $x^2 = \Pi_{i \in S} a_i^2$. If the a_i are chosen at random [4], there is a 50% chance (or more if n is the product of more than two primes) that $x \not\equiv \pm y \pmod n$. If x is congruent to $\pm y$, one simply finds another pair a_{k+2}, b_{k+2}, and another linear dependency, and tries again until n is factored.

For CF, one searches for the a_i's among the numerators of the continued fraction convergents of $\sqrt{n}$. To test whether a candidate for b_i can be factored over the factor base, one must try to divide it by each prime p_j. In CFEA, these trial divisions are aborted if the candidate fails too many of the early ones.

For QS, one searches for the a_i's among consecutive integers starting with $[\sqrt{n}]$, the integral part of $\sqrt{n}$. It is only necessary to divide n and $[\sqrt{n}]$ by each p_j and solve a quadratic congruence mod p_j. After that, large blocks of possible b_i's can be sieved at once, and no trial divisions are needed.

III. Implementation of the Quadratic Sieve. The quadratic sieve algorithm was implemented with a FORTRAN program on an HP3000/series 3 computer. The initial factor base contained 1000 elements, namely the first 999 primes modulo which n (the number factored) is a quadratic residue, plus the "prime" -1. The element -1 was included so that the numbers b_i could be negative as well as positive, thus doubling the number of candidates for b_i less than a given absolute value.

First, the quadratic congruence

$$\left(\left[\sqrt{n}\right] + z\right)^2 - n \equiv 0 \pmod p$$

was solved for p equal to each prime in the factor base, and also modulo the squares of all the primes, the cubes of all primes less than 130, and higher powers of the smallest primes. This process required 7 minutes of CPU time, using a fast algorithm in the spirit of Lehmer [5].

The integers $[\sqrt{n}] + m$ were then sieved, in blocks of 10000, for m up to 400,000,000 and down to $-499{,}999{,}999$. Each block was sieved by initially setting each element of a 10000 element array, SUMLOG, to the logarithm of the corresponding integer in each block. Then, for each odd prime p in the factor base, the logarithm of p was subtracted from SUMLOG(J), where J was initially set to z_1 and z_2, respectively, the two solutions of the above congruence, and then increased by increments of p until J was greater than 10000. The final values of J were then decreased by 10000 and stored, to be used as initial values for the next block. This procedure was then repeated for p^2, and, where appropriate, for higher powers of p. For powers of p too high to be included in the sieve, trial division was employed where necessary. The same procedure was then used for powers of two, except that J was incremented by the previous power of two, since in this case $n \equiv 1 \pmod 8$, and therefore the quadratic congruences modulo powers of two have four solutions each. To speed things up by using fixed-point instead of floating-point arithmetic, and to conserve memory, all logarithms were taken to be the integral part of $10 \log_2$.

After sieving with all primes and prime powers, the array SUMLOG was scanned to pick out values close to zero, which would indicate a number $b_i = ([\sqrt{n}] + m)^2 - n$

completely factored into primes in the factor base. For each such b_t, the vector $\vec{v}_t$ was computed and stored, along with m, in a disc file. Altogether, 327 such numbers were found. In addition, SUMLOG was scanned to pick out values less than twice the log of p_{1000}, the largest prime in the factor base. These elements of SUMLOG corresponded to numbers b_t which factored into primes in the factor base, except for one larger factor which was less than p_{1000}^2, and hence prime. The vectors for these numbers, 25747 in all, were also stored in a disc file, along with the large prime factor of each. The file was then searched for pairs of numbers with the same large prime factor, and 690 such pairs were found (including several triplets, each counted as two pairs).

A total of 70 hours of CPU time was required for the sieving, plus another 8 minutes to find the 690 matching pairs. Very little of the 70 hours (less than one hour) was spent processing numbers b_t after they had been found.

Each component v_{tj} of the vectors was stored as a single bit; thus sixteen components were stored in each 16-bit word, and 63 words sufficed for each vector. As each matching pair of numbers was found, the two vectors were added together to cancel out the common large prime factor. The 690 vector s thus formed were combined with the 327 vectors of the completely factored numbers to form a 1017×1000 matrix. This matrix was then reduced by Gaussian elimination, and 27 different linear dependencies were found. All vector additions were done using the logical operation XOR (equivalent to addition in the integers mod 2) so 16 pairs of components could be added in a single operation. Despite the packing of the matrix into 1017×63 sixteen-bit words, the entire matrix could not fit into the 32767 word maximum data stack of the HP3000. The matrix reduction routine therefore used virtual memory, with no more than 208 rows of the matrix in the core memory at any time.

The reduction of the matrix required 340 seconds of CPU time. Of this time, approximately 90 seconds were spent on row operations, and the rest on reading and writing to the disc file and searching for rows with a 1 in a given column. The distinction is important because the time required for row operations is proportional to k^3 (where k is the number of elements in the factor base) while the time required for the other procedures is proportional to k^2. (If the core memory is held fixed, then the time required for reading and writing also grows as k^3, but we shall show in Section V that with an efficient program this time is small compared to the time for row operations.)

Once the matrix was reduced, a few more minutes were required to factor n. Fortunately it was not necessary to calculate x and y (which would have taken several hours, since these numbers have thousands of digits), but only their residues (mod n). The residue of x was computed by taking the product of all a_t's in $Z \bmod n$. To compute the residue of y, each b_i was represented as the formal product of primes, and these primes were combined to yield y^2 as a formal product. Each exponent of this formal product was then divided by two, and the prime factors multiplied together in $Z \bmod n$.

IV. Comparison of Running Times. Let

$$L = e^{\sqrt{\log n \log\log n}}.$$

For all factoring algorithms of the kind discussed in this paper, the time required to factor n is believed to be a function of the form

$$L^{\alpha+o(1)},$$

where α is a constant and $o(1) \to 0$ as $n \to \infty$.

For each algorithm, this belief can be proved [7] and the value of α computed if we assume that the special numbers among which one searches for the b_i's are just as likely to factor into small primes as random numbers of the same order of magnitude. For CF, $\alpha = \sqrt{2}$, for CFEA, $\alpha = \sqrt{3}/\sqrt{2}$, and for QS, $\alpha = 3\sqrt{2}/4$ (this last value can be improved slightly by using the Coppersmith-Winograd method [3] to reduce the matrix, but this is not practical for reasonable values of n).

The optimum size, k, for the factor base is

$$L^{\beta+o(1)},$$

where $\beta = \sqrt{2}/4$ for both CF and QS, and $1/\sqrt{6}$ for CFEA.

Both α and β are asymptotic values. However, one would expect QS to be slower than CF and CFEA for small n for the following reason: Let a/c be a continued fraction convergent to $\sqrt{n}$, and let $b = a^2 - c^2 n$. Then $b = O(\sqrt{n})$, and the probability that b can be factored over the factor base is $u^{-u+o(u)}$, where

$$u = \log \sqrt{n} / \log p_k$$

(p_k being the largest prime on the factor base) [2]. To find k such numbers, one would expect to have to search

$$M = ku^{u+o(u)}$$

continued fraction convergents. On the other hand, if we use QS, with $a = [\sqrt{n}] + m$ and $b = a^2 - n$, then $b = O(m\sqrt{n})$, and

$$u = \left(\log m + \log \sqrt{n}\right)/\log p_k.$$

Typically, m is on the order of M, the total number of numbers sieved, so we can estimate M by simultaneously solving the two equations $u = (\log M + \log \sqrt{n})/\log p_k$ and $M = ku^u$. For very large n, the term $\log M$ becomes unimportant in that it does not affect the value of α or β. However, for $n \approx 10^{47}$, $\log\sqrt{n} \approx 23.5 \log 10$ and, even for CF and CFEA, M is typically [9] around 2×10^7 (slightly larger if n is not a quadratic residue mod 3, 5, and 8), so $\log M \approx 7 \log 10$, a significant fraction of $\log\sqrt{n}$ Hence u is significantly larger for QS than for CF and CFEA and M must also be larger; as we have seen it is just under 9×10^8 in this instance. We save time by testing numbers with a sieve instead of trial division, but for small n we lose more time than we save because we have more numbers to test (see also Section 9 of [7]).

Furthermore, to minimize the running time of QS, we should choose k so that the sieving time is on the same order as the matrix reduction time. Thus any increase in the sieving time implies an increase in the optimum size of k. We would therefore expect k to be larger for QS than for CF although both algorithms have the same value of β. Empirical data on this question is not available for CF, since Wagstaff [9] used only one value of k, viz. 223. For QS, $k = 1000$ is too small, since the sieving time (70 hours) is much greater than the matrix reduction time (5 minutes) with this value of k.

To estimate the optimum value of k for QS, let

$$k = L^{\sqrt{2}/4+\delta},$$

where $|\delta|$ is small. Then from [7] we have

$$M = L^{\sqrt{2}/4+\delta+(\sqrt{2}+4\delta)^{-1}+o(1)}.$$

Note that

$$\frac{1}{\sqrt{2}+4\delta} = \frac{1}{\sqrt{2}} \cdot \frac{1}{1+4\delta/\sqrt{2}} = \frac{1}{\sqrt{2}}\left(1 - \frac{4\delta}{\sqrt{2}} + O(\delta^2)\right) = \frac{1}{\sqrt{2}} - 2\delta + O(\delta^2),$$

so that

$$kM = L^{\sqrt{2}+O(\delta^2)+o(1)}.$$

Thus for δ near 0, we have kM approximately constant, provided the error term $o(1)$ does not behave too wildly. The time required to sieve a given interval only grows as

$$\sum_{j=1}^{k} \frac{1}{p_j} = O(\log\log k).$$

Therefore the sieving time should be inversely proportional to k. The matrix reduction time, with Gaussian elimination, is proportional to k^3. Therefore, to minimize the total time, the sieving time should be about three times the matrix reduction time. If we naively extrapolate from $k = 1000$, we obtain an optimum value of $k = 5000$, with a sieving time of $70/5 = 14$ hours, and a matrix reduction time of $1.5 \times 5^3 + 4 \times 5^2$ minutes $= 5$ hours.

I attempted to implement the quadratic sieve algorithm with a factor base of 4700 primes, but it soon became clear that this was not practical on the HP3000 without a major modification of the program. For one thing, the data stack was not large enough to hold all 4700 primes, along with a logarithm and both solutions of a quadratic congruence for each prime. It was therefore necessary to use virtual memory, and this increased the sieving time by a factor of three, although it should be possible to write a more efficient virtual memory program, as we shall see. It was also necessary to ignore those b_t which had a large prime factor (the "large prime variation") and include only those which factored entirely over the factor base; otherwise too much time would have been spent processing the numbers b_t after they had been found. This increased the sieving time by another factor of three over the naive extrapolation discussed above. However, I was able to confirm empirically that M varies inversely with k, since three of the first 10^6 numbers sieved were completely factored over a factor base of 1000, while eight of the first 10^5 numbers factored over a factor base of 4700.

It would appear that if core memory was not an obstacle, or a more efficient virtual memory program was employed, k should be about 7000 for n on the order of 10^{47}. This would give a sieving time of about 30 hours (since the large prime variation cannot be used) and a matrix reduction time of 12 hours. Both times would likely be somewhat less, the former because the numbers being sieved would be smaller on the average than they are for $k = 1000$ (and therefore more likely to factor), and the latter because the matrix would be sparser. We will assume a total running time of 40 hours.

Wagstaff's [9] CF and CFEA programs are able to factor comparable numbers in about 30 hours and 5 hours, respectively, on an IBM370/158. A short test program revealed that this machine is about twice as fast as the HP3000 for the QS program, making CF about two-thirds as fast as QS, and CFEA four times faster. However, Wagstaff's programs were partly in assembly language, while mine were entirely in FORTRAN, so the true ratio might be closer to 0.5 for QS/CF and 3 for QS/CFEA.

To estimate those values of n where the running times of QS and CF (resp. CFEA) are equal, we must first estimate how fast CF/QS and CFEA/QS change with n. Asymptotically, $\alpha = 1.061$ for QS, 1.414 for CF, and 1.225 for CFEA. However, all three exponents are somewhat lower for n on the order of 10^{40} to 10^{70}. Indeed, for CF we can calculate the running time directly for any value of n by choosing k so as to minimize k^2u^u. These calculations reveal that α is closer to 1.2 for n in this range, and this is confirmed by Wunderlich's observation [10] that the running time for CF grows approximately as $n^{1/7}$. For QS, we can start with the fact that k should be about 7000 for $n = 10^{47}$, and compute optimum values of k for larger n by requiring k^3 and ku^u to maintain a constant ratio. These calculations give $\alpha = 0.9$. Therefore the running time of CF/QS grows as $L^{0.3}$. For $n = 10^{47}$, $L \approx e^{22.5}$, so the running times of CF and QS should be equal when $L = \exp[22.5 + (\log 0.5)/0.3] = e^{20.2}$ and $n = 10^{39}$. This should be increased, perhaps to 10^{40}, because for CF the large prime variation is slightly more efficient with smaller n, while for QS it is still useless since k is around 4000.

For CFEA, α depends on the number of early aborts; 1.225 is the limit for an infinite number. Wagstaff [9] estimates that the optimum number of early aborts for this range of n is three, which would give an asymptotic value of $\sqrt{13/8} = 1.275$ for α [7], and an actual value (by analogy with CF and QS) of about 1.1. This would imply that CFEA/QS grows as $L^{0.2}$, with the running times equal at $L = \exp[22.5 + (\log 3)/0.2] = e^{28}$ and $n = 10^{68}$. In this case the large prime variation is less efficient for CFEA, perhaps by a factor of 1.5, so L should probably be about $\exp[22.5 + (\log 2)/0.2] = e^{26}$ and n about 10^{60}. A 60-digit number would take about 300 hours to factor on an IBM370/158.

Both CFEA and QS should be able to take full advantage of vector processing, CFEA by doing many trial divisions at once, and QS by sieving many consecutive subintervals at once, the length of the subintervals being divisible by p_j. This suggests that QS might be a practical method for factoring 70 or 80-digit numbers on a Cray 1 or similar machine. However, the 64-bit word size of the Cray 1 favors CFEA, since trial divisions are faster while sieving is not affected, so the break-even point between QS and CFEA might be closer to 70 digits.

Imagine a super-computer, one million times as fast as an IBM370/158 (Josephson junctions, pipelining, and a 1000-element vector processor). On such a machine, a 100-digit number could be factored in about three days with QS, or three weeks with CFEA.

V. Fast Virtual Memory Techniques. All three algorithms require k^2 bits of memory to store the matrix. This requirement cannot be reduced much by sparse matrix techniques because the matrix becomes denser as it is reduced. Indeed, for

the QS factorization described here, the matrix became saturated (half ones and half zeroes) after about 600 of the original 1000 columns had been zeroed out. But as long as the core memory is large enough to hold many rows of the matrix, the rest of the matrix can be stored in a disc file, and comparatively little time will be spent transferring rows between the disc and the core.

If, however, k begins to approach or exceed the size of the core, then more care must be taken. For example, if at any given time the core contains elements from only two different rows, then the time spent reading from the disc will be of the same order of magnitude as the time spent adding rows together. This difficulty can be eliminated by partitioning the matrix into $r \times r$ square pieces (i.e. each square is k/r elements on a side) and storing two such squares from the same column in the core at any given time. In addition, the core must contain a third square to keep track of which rows of the first square are to be added to each row of the second square. Since there are $O(r^2)$ different combinations of two squares from each of the r columns of squares, a total of $O(r^3)$ squares will be read from the disc, and, since each square contains $(k/r)^2$ elements, this will require a total of $O(rk^2)$ READ operations. But the total number of additions required to reduce the matrix is on the order of k^3, so a neglible time will be spent reading from the disc.

For CF and CFEA, there are no other significant memory requirements, since only $O(\log n)$ words of core are required to perform a trial division. The k elements of the factor base can be stored in a disc file without slowing things significantly, because much more time is required to do a trial division than to read one prime.

With QS, on the other hand, special precautions must be taken when k is comparable to or greater than the number of words in the core memory. The time required to sieve an interval of length s is proportional to $s \log\log k$, but in addition to the sieving process itself, certain bookkeeping routines must be performed on each prime, requiring time proportional to k. If k is on the order of s, then a significant fraction of the time is spent on these bookkeeping routines. If s is fixed and k is allowed to grow much larger than s, then the time required to check whether each number can be factored will be proportional to k, just as in CF. It follows that α will increase to $\sqrt{2}$. This problem can be avoided by only performing the bookkeeping routines on a small fraction of the primes in the factor base, namely those primes which divide at least one number in the interval being sieved. Let $q = p_k/s$, rounded up to the nearest integer. Those primes less than s would always remain in the core, while the larger primes (two copies of each, one for each solution of the quadratic congruence) would be stored in q different disc files. For each interval sieved, only the primes in one disc file would be used, with consecutive files being read (in a wraparound sequence) for consecutive intervals. After each prime p is used, it would be written into one of the other disc files, namely the file r places ahead in the sequence, where r is the integral part of $(m + p)/s$, and the mth element of the interval being sieved is divisible by p.

Department of Mathematics
Rutgers University
Camden, New Jersey 08102

1. J. BRILLHART, D. H. LEHMER, J. L. SELFRIDGE, B. TUCKERMAN & S. S. WAGSTAFF, Jr., "Factorizations of $b^n \pm 1$ up to high powers." (To appear).

2. E. R. CANFIELD, P. ERDOS & C. POMERANCE, "On a problem of Oppenheim concerning 'Factorisatio Numerorum'," *J. Number Theory*. (To appear.)

3. D. COPPERSMITH & S. WINOGRAD, "On the asymptotic complexity of matrix multiplication," *SIAM J. Comput.* (To appear.)

4. J. D. DIXON, "Asymptotically fast factorization of integers," *Math. Comp.*, v. 36, 1981, pp. 255–260.

5. D. H. LEHMER, "Computer technology applied to the theory of numbers," in *Studies in Number Theory* (W. J. LeVeque, ed.), Math. Assoc. Amer., 1969, pp. 117–151.

6. M. A. MORRISON & J. BRILLHART, "A method of factoring and the factorization of F_7," *Math. Comp.*, v. 29, 1975, pp. 183–205.

7. C. POMERANCE, "Analysis and comparison of some factoring algorithms," in *Computational Methods in Number Theory* (H. W. Lenstra, Jr. and R. Tijdeman, eds.), Math. Centrum, Amsterdam. (To appear.)

8. C. P. SCHNORR, private correspondence dated 1982 (communicated to the author by C. Pomerance).

9. S. S. WAGSTAFF, JR., private correspondence dated 1981–1982.

10. M. C. WUNDERLICH, "A running time analysis of Brillhart's continued fraction factoring method," in *Number Theory Carbondale* 1979 (M. B. Nathanson, ed.), Lecture Notes in Math., Vol. 751, Springer-Verlag,1979, pp. 328–342.