CIRCUITS FOR INTEGER FACTORIZATION: A PROPOSAL

DANIEL J. BERNSTEIN

Abstract. The number field sieve takes time $L^{1.901\cdots+o(1)}$ on a general-purpose computer with $L^{0.950\cdots+o(1)}$ bits of memory; here L is a particular subexponential function of the input size. It takes the same time on a parallel trial-division machine, such as Cracker or TWINKLE, of size $L^{0.950\cdots+o(1)}$. It takes time only $L^{1.185\cdots+o(1)}$ on a machine of size $L^{0.790\cdots+o(1)}$ explained in this paper. This reduction of total cost from $L^{2.852\cdots+o(1)}$ to $L^{1.976\cdots+o(1)}$ means that a $((3.009\cdots+o(1))d)$-digit factorization with the new machine has the same cost as a d-digit factorization with previous machines.

0. Preface

This paper is an excerpt from a grant proposal that I submitted to NSF DMS at the beginning of October 2001.

The same techniques can be applied to other congruence-combination algorithms for factoring, discrete logarithms, class groups, etc. See [3] for a bibliography.

Priority dates. I realized on 13 September 2000 that special-purpose hardware would change the exponent in the cost of integer factorization. I announced the exponent reduction from $3+o(1)$ to $2.5+o(1)$ for real (two-dimensional) circuits in a seminar at Butler University on 23 March 2001, a rump-session presentation at Eurocrypt 2001 on 7 May 2001, and a talk at the Algorithms and Number Theory conference at Dagstuhl on 14 May 2001. I realized on 9 August 2001 that the sieving exponent could easily be reduced from $2.5+o(1)$ to $2+o(1)$.

1. Introduction

It is conjectured that one can find the prime factors of an integer n in time $L^{O(1)}$, where $L=\exp((\log n)^{1/3}(\log\log n)^{2/3})$.

More precisely: Write $c=(92+26\sqrt{13})^{1/3}$. The number field sieve, with sensibly chosen parameters, takes time $L^{c/3+o(1)}=L^{1.9018836118\cdots+o(1)}$ on a general-purpose computer with $L^{c/6+o(1)}=L^{0.9509418059\cdots+o(1)}$ bits of memory, and is conjectured to find the prime factors of n.

I realized recently that the same computation can be carried out in time only $L^{c/4+o(1)}=L^{1.4264127088\cdots+o(1)}$ on a different machine of size $L^{c/6+o(1)}$. Another parameter choice takes time $L^{d+o(1)}=L^{1.1856311014\cdots+o(1)}$ on a machine of size $L^{2d/3+o(1)}=L^{0.7904207343\cdots+o(1)}$, and is still conjectured to find the prime factors of n. Here $d=(5/3)^{1/3}$.

Date: 20011109.

1991 *Mathematics Subject Classification.* Primary 11Y05. Secondary 68W10.

The author was supported by the National Science Foundation under grant DMS-9970409.

The cost of factorization—the product of the time and the cost of the machine—has thus dropped from $L^{c/2+o(1)} = L^{2.8528254177\cdots+o(1)}$ to

$$L^{5d/3+o(1)} = L^{1.9760518358\cdots+o(1)}.$$

In other words, for a given cost, the number of digits of n has grown by a factor of $(3c/10d + o(1))^3 = 3.0090581972\cdots + o(1)$.

This is a tremendously exciting observation; it demands further investigation. What do all the $o(1)$'s look like in practice? Are these machines more cost-effective than general-purpose computers for current ranges of n? See sections 2 through 6 of this proposal.

A team led by Herman te Riele used the number field sieve on general-purpose computers to factor a difficult 512-bit integer in August 1999. Is it now possible to factor 1536-bit integers at reasonable cost?

2. Odd-even transposition sorting

Odd-even transposition sorting is a straightforward algorithm that sorts m numbers in m steps on a one-dimensional machine of size m. Readers familiar with the algorithm may skip to the next section; this section is purely expository.

The machine has m cells, each cell holding one number, each cell connected to the adjacent cells. In the first step, the first and second cells sort their two numbers; the third and fourth cells sort their two numbers; etc. In the second step, the second and third cells sort their two numbers; the fourth and fifth cells sort their two numbers; etc. The third step is just like the first step; the fourth step is just like the second step; and so on.

There are several ways to prove that m steps suffice to sort the entire list of numbers. See, e.g., [7, exercise 5.3.4–37].

The following table is an example of odd-even transposition sorting, with $m = 8$:

Time 0:	8	9	7	9	3	2	3	4
Time 1:	8	9	7	9	2	3	3	4
Time 2:	8	7	9	2	9	3	3	4
Time 3:	7	8	2	9	3	9	3	4
Time 4:	7	2	8	3	9	3	9	4
Time 5:	2	7	3	8	3	9	4	9
Time 6:	2	3	7	3	8	4	9	9
Time 7:	2	3	3	7	4	8	9	9
Time 8:	2	3	3	4	7	8	9	9

The notation $\frac{a\ b}{c\ d}$ means $c = \min\{a, b\}$ and $d = \max\{a, b\}$.

3. Schimmler sorting

Schimmler's algorithm sorts m^2 numbers in $8m - 8$ steps on a two-dimensional machine of size m^2, when m is a power of 2.

The machine consists of m^2 cells in an $m \times m$ mesh, each cell holding one number, each cell connected to the adjacent cells. There are several natural orderings of cells in an $m \times m$ mesh. Schimmler's algorithm can sort using the left-to-right order

$$(1,1),(1,2),\ldots,(1,m),(2,1),(2,2),\ldots,(2,m),(3,1),(3,2),\ldots,(3,m),\ldots;$$

the right-to-left order

$$(1,m),(1,m-1),\ldots,(1,1),(2,m),(2,m-1),\ldots,(2,1),\\(3,m),(3,m-1),\ldots,(3,1),\ldots;$$

or the snakelike order

$$(1,1),(1,2),\ldots,(1,m),(2,m),(2,m-1),\ldots,(2,1),(3,1),(3,2),\ldots,(3,m),\ldots.$$

Schimmler's algorithm works as follows. Recursively sort the top-left quadrant of the mesh, left to right; the top-right quadrant of the mesh, left to right; the bottom-left quadrant of the mesh, right to left; and the bottom-right quadrant of the mesh, right to left. Sort each column independently, top to bottom, with odd-even transposition sort. Sort each row independently, snakelike. Sort each column independently, top to bottom. Finally, sort each row independently, using the desired order, left to right or right to left or snakelike.

For example, take the following array:

3	1	4	1	5	9	2	6
5	3	5	8	9	7	9	3
2	3	8	4	6	2	6	4
3	3	8	3	2	7	9	5
0	2	8	8	4	1	9	7
1	6	9	3	9	9	3	7
5	1	0	5	8	2	0	9
7	4	9	4	4	5	9	2

Sort the quadrants:

1	1	2	3	2	2	2	3
3	3	3	3	4	5	5	6
3	4	4	5	6	6	7	7
5	8	8	8	9	9	9	9
1	1	0	0	2	2	1	0
4	4	3	2	5	4	4	3
7	6	5	5	9	8	7	7
9	9	8	8	9	9	9	9

Sort the columns, top to bottom:

1	1	0	0	2	2	1	0
1	1	2	2	2	2	2	3
3	3	3	3	4	4	4	3
3	4	3	3	5	5	5	6
4	4	4	5	6	6	7	7
5	6	5	5	9	8	7	7
7	8	8	8	9	9	9	9
9	9	8	8	9	9	9	9

Sort the rows, snakelike:

$$\begin{matrix} 0&0&0&1&1&1&2&2\\ 3&2&2&2&2&2&1&1\\ 3&3&3&3&3&4&4&4\\ 6&5&5&5&4&3&3&3\\ 4&4&4&5&6&6&7&7\\ 9&8&7&7&6&5&5&5\\ 7&8&8&8&9&9&9&9\\ 9&9&9&9&9&9&8&8 \end{matrix}$$

Sort the columns, top to bottom:

$$\begin{matrix} 0&0&0&1&1&1&1&1\\ 3&2&2&2&2&2&2&2\\ 3&3&3&3&3&3&3&3\\ 4&4&4&5&4&4&4&4\\ 6&5&5&5&6&5&5&5\\ 7&8&7&7&6&6&7&7\\ 9&8&8&8&9&9&8&8\\ 9&9&9&9&9&9&9&9 \end{matrix}$$

Sort the rows, left to right:

$$\begin{matrix} 0&0&0&1&1&1&1&1\\ 2&2&2&2&2&2&2&3\\ 3&3&3&3&3&3&3&3\\ 4&4&4&4&4&4&4&5\\ 5&5&5&5&5&5&6&6\\ 6&6&7&7&7&7&7&8\\ 8&8&8&8&8&9&9&9\\ 9&9&9&9&9&9&9&9 \end{matrix}$$

The array is now sorted, left to right.

To prove the correctness of an algorithm of this type, select a cutoff value v, and consider the positions of numbers larger than v. After the recursive sorting, all that matters is how many such numbers are in each quadrant. It is then easy to analyze where those numbers appear in subsequent steps.

History. Thompson and Kung in [21] showed that an $m \times m$ mesh can sort m^2 numbers in $O(m)$ steps. Schnorr and Shamir in [18] showed that an $m \times m$ mesh can sort m^2 numbers in snakelike order in $(3 + o(1))m$ steps. Schimmler's algorithm appeared in [17]; it is considerably simpler than the Schnorr-Shamir algorithm, although it is not as fast.

Similar comments apply to higher-dimensional meshes. Unfortunately, it is difficult in practice to build an $m \times m \times m$ mesh for large m.

A philosophical note. I always thought that common general-purpose computers were the pinnacle of realistic computational power. Special-purpose computer architectures, such as Lehmer's bicycle chain sieve or Pomerance's Cracker or Shamir's TWINKLE, were at best a constant factor faster. Quantum computers are asymptotically faster for many computations, but it is unclear whether they can actually be built.

I also thought that parallel computing reduced the *time*, not the *cost*, of computations. Ten processors might perform a computation in one tenth the time of a single processor, but they are ten times as expensive, so the cost of the computation remains the same.

I was wrong. Schimmler's machine, with m^2 processors, can be built for $m^{2+o(1)}$ dollars, just like a single-processor computer with $m^{2+o(1)}$ bits of memory. It can sort m^2 numbers in time $m^{1+o(1)}$, while the single-processor machine needs time $m^{2+o(1)}$. The cost of the computation has dropped from $m^{4+o(1)}$ to $m^{3+o(1)}$.

4. Circuits for linear algebra

Let A be a square matrix over $\mathbf{F}_2$ with $y^{1+o(1)}$ columns and with $y^{o(1)}$ nonzero entries in each column. The obvious method of computing Av, given a vector v over $\mathbf{F}_2$, takes time $y^{1+o(1)}$ on a general-purpose computer with $y^{1+o(1)}$ bits of memory.

One can do better with Schimmler sorting: time $y^{0.5+o(1)}$ on another machine of size $y^{1+o(1)}$. In particular, this machine can compute a dot product in time $y^{0.5+o(1)}$. Here are the details.

Select $m \in y^{0.5+o(1)}$ as a power of 2 large enough that m^2 exceeds the number of nonzero entries of A plus twice the number of rows of A. Build an $m \times m$ mesh of cells, each cell having $O(\log y)$ bits of storage.

Store the nonzero entries of v—the integers j such that $v_j = 1$—in these cells in any order. Also store the nonzero entries of M—the pairs (i, j) such that $M_{i,j} = 1$—in cells in any order; note that there are only $y^{o(1)}$ pairs for each j. Store 0 in all remaining cells.

Sort all the integers j and pairs (i, j) in order of j, with the cells in snakelike order. For example:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
$(3,3)$	$(2,3)$	$(3,2)$	$(2,1)$	$(1,1)$	1	0	0
$(8,3)$	5	$(1,5)$	$(4,5)$	6	7	$(6,7)$	$(15,7)$
$(1,12)$	12	$(13,11)$	$(1,10)$	$(1,9)$	$(8,8)$	$(2,8)$	8
$(10,12)$	13	$(1,13)$	$(2,13)$	14	$(1,14)$	$(3,14)$	$(4,14)$
$(11,16)$	$(3,16)$	$(2,16)$	16	$(5,15)$	$(4,15)$	$(2,15)$	$(1,15)$

This takes $m^{1+o(1)} = y^{0.5+o(1)}$ steps, and brings each j within distance $y^{o(1)}$ of all the cells with pairs (i, j). Communicate each j to those cells; this takes $y^{o(1)}$ steps. Then replace the j's by new numbers: i in a cell that has both j and (i, j); 0 in all other cells. For example:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
$(3,3)$	$(2,3)$	$(3,2)$	$2,(2,1)$	$1,(1,1)$	0	0	0
$(8,3)$	0	$1,(1,5)$	$4,(4,5)$	0	0	$6,(6,7)$	$15,(15,7)$
$1,(1,12)$	0	$(13,11)$	$(1,10)$	$(1,9)$	$8,(8,8)$	$2,(2,8)$	0
$10,(10,12)$	0	$1,(1,13)$	$2,(2,13)$	0	$1,(1,14)$	$3,(3,14)$	$4,(4,14)$
$11,(11,16)$	$3,(3,16)$	$2,(2,16)$	0	$(5,15)$	$(4,15)$	$(2,15)$	$(1,15)$

Sort (snakelike) these new numbers; this takes $y^{0.5+o(1)}$ steps. For example:

1	1	1	1	1	2	2	2
10	8	6	4	4	3	3	2
11	15	0	0	0	0	0	0
$(3,3)$	$(2,3)$	$(3,2)$	$(2,1)$	$(1,1)$	0	0	0
$(8,3)$	0	$(1,5)$	$(4,5)$	0	0	$(6,7)$	$(15,7)$
$(1,12)$	0	$(13,11)$	$(1,10)$	$(1,9)$	$(8,8)$	$(2,8)$	0
$(10,12)$	0	$(1,13)$	$(2,13)$	0	$(1,14)$	$(3,14)$	$(4,14)$
$(11,16)$	$(3,16)$	$(2,16)$	0	$(5,15)$	$(4,15)$	$(2,15)$	$(1,15)$

Compare each cell to one of its two (snakelike) neighbors, as in the first step of an odd-even transposition sort; if the two cells have the same number i, replace that number by 0 in both cells. Then sort once more. For example:

1	2	2	3	3	4	4	6
0	0	0	0	15	11	10	8
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	$(1,1)$	$(2,1)$	$(3,2)$	$(2,3)$	$(3,3)$
$(1,9)$	$(8,8)$	$(2,8)$	$(15,7)$	$(6,7)$	$(4,5)$	$(1,5)$	$(8,3)$
$(1,10)$	$(13,11)$	$(1,12)$	$(10,12)$	$(1,13)$	$(2,13)$	$(1,14)$	$(3,14)$
$(11,16)$	$(3,16)$	$(2,16)$	$(5,15)$	$(4,15)$	$(2,15)$	$(1,15)$	$(4,14)$

At this point there are at most two occurrences of each i. Compare each cell to both of its neighbors, and cancel any remaining duplicates. That's it. The nonzero entries of v and M have been replaced by the nonzero entries of Mv and M.

Computing kernels. Wiedemann's algorithm in [22] computes the minimal polynomial f of A as follows.

Select uniform random vectors u and v. The minimal polynomial g of the bit sequence $uv, uAv, uA^2v, \ldots$ is a divisor of f. The least common multiple of a few such divisors is, with high probability, f.

One can compute g very quickly from the first $y^{1+o(1)}$ bits in the sequence. The algorithms in [19], [13], [4], and [14], with the help of fast multiplication, do this in time $y^{1+o(1)}$ on a general-purpose computer with $y^{1+o(1)}$ bits of memory.

The obvious method of computing these $y^{1+o(1)}$ bits, multiplying v by A repeatedly and multiplying each result by u, takes time $y^{2+o(1)}$ on the same computer. It takes time only $y^{1.5+o(1)}$ on the machine described above.

Given the minimal polynomial of A, one can easily construct random elements of the kernel of A. The obvious method again takes time $y^{2+o(1)}$ on a general-purpose computer with $y^{1+o(1)}$ bits of memory; the machine described above takes time $y^{1.5+o(1)}$.

Plans. I will investigate the cost of these computations in detail. Exactly how expensive are linear-algebra circuits of various sizes? Computer programs are available to help construct and simulate dedicated circuits and FPGAs, producing precise measurements of size and speed.

Are there better representations of matrices and vectors? For example, should j and (i, j) be assigned permanently to cells? Should the third sorting step be eliminated? Can repeated i's be profitably removed in the middle of Schimmler's algorithm? Is it practical to use the Schnorr-Shamir algorithm instead of Schimmler's algorithm? There is a huge literature on mesh routing and mesh sorting, with dozens of potentially useful techniques.

There are many more ways to save constant factors. Wiedemann's algorithm can handle additional pairs (u, v) much more quickly once a large divisor of f is known. One can use Lanczos-type algorithms instead of Wiedemann's algorithm; see [8] for a survey. I will explore all of these possibilities.

A block version of Wiedemann's algorithm allows further parallelization, although it does not change the cost of the computation. See [6]. It should be possible to combine $y^{0.1+o(1)}$ of these machines, for example, to construct random elements of the kernel of A in time $y^{1.4+o(1)}$.

5. Circuits to find smooth numbers

Consider a set of $y^{2+o(1)}$ numbers, each with $(\log y)^{O(1)}$ digits. How long does it take to find all the y-smooth numbers?

RAM sieving. Common practice is to partition the set into $y^{1+o(1)}$ pieces, each of size $y^{1+o(1)}$, and sieve each piece. See [3] for a method that achieves similar performance even if the numbers are not sieveable.

Sieving seems very efficient. It handles $y^{2+o(1)}$ numbers in $y^{2+o(1)}$ steps. However, it requires $y^{1+o(1)}$ bits of memory, only a few of which are performing productive work at any moment. Most of the bits are simply sitting around, twiddling their thumbs. The *cost* of sieving is $y^{3+o(1)}$.

Parallel trial division. Another approach is to divide each of the $y^{2+o(1)}$ numbers by each of the $y^{1+o(1)}$ primes.

This may seem slower than sieving: it takes $y^{3+o(1)}$ steps. However, it uses only $y^{o(1)}$ bits of memory, so it can easily be parallelized. One can handle separate numbers in parallel, or handle separate primes in parallel, or both. One can also speed up the trial division by a factor of $y^{o(1)}$ when the numbers are sieveable.

The *cost* of any of these approaches is $y^{3+o(1)}$: in other words, within a factor $y^{o(1)}$ of the cost of sieving. This applies, in particular, to Pomerance's Cracker, and Shamir's TWINKLE.

Parallel ECM. Trial division is not the state of the art in low-memory smoothness-testing methods. ECM, Lenstra's elliptic-curve method in [10], has conjecturally negligible chance of error, and takes time at most $\exp\sqrt{(2+o(1))\log y \log\log y}$ per integer. HECM, the Lenstra-Pila-Pomerance hyperelliptic-curve method in [11], has provably negligible chance of error, and takes time at most $\exp((\log y)^{2/3+o(1)})$ per integer. Both methods use $y^{o(1)}$ bits of memory.

Consequently a parallel ECM or HECM machine, handling $y^{1+o(1)}$ numbers in parallel, has size $y^{1+o(1)}$, and tests smoothness of $y^{2+o(1)}$ numbers in time $y^{1+o(1)}$. The *cost* of this computation is only $y^{2+o(1)}$.

Note that numbers are handled by this machine much more quickly than they could be communicated through a serial link. This machine is not useful unless it receives inputs in parallel. If there are many outputs then the outputs also need to be handled in parallel.

Plans. As in section 4, I will investigate the cost of these computations in detail.

There are several ways to achieve cost $y^{2.5+o(1)}$: parallel Pollard rho, for example, or sieving via Schimmler's algorithm. These methods may be faster than ECM for current values of y, and can be profitably used as a first step in any case.

There are many more options to explore. For example, as shown by Pomerance in [15], early aborts discard a sizable fraction of useful inputs, but reduce the time by a larger fraction, when the abort parameters are chosen properly.

6. Circuits for integer factorization

The number field sieve tries to factor an integer $n \geq 15$ as follows, when n is odd and not a prime power. The specific parameter choices here are due to Coppersmith in [5].

Define $\alpha = (\log n)^{1/3}(\log\log n)^{-1/3}$ and

$$L = n^{1/\alpha^2} = \exp((\log n)^{1/3}(\log\log n)^{2/3}).$$

Note that $(1 + o(1))\alpha \log \alpha = (1/3 + o(1)) \log L$.

Select an integer degree $d \in (1.4017532352\cdots + o(1))\alpha$ with $d \geq 2$. The constant here is $(92 + 26\sqrt{13})^{1/3}(-5 + 2\sqrt{13})/9$.

Select an integer m close to $n^{1/d}$. Write n as $m^d + f_{d-1}m^{d-1} + \cdots + f_1 m + f_0$ with each f_i bounded by $n^{(1+o(1))/d}$. There are some bad choices of f_i's that will make the rest of the algorithm fail, but a random choice is conjectured to succeed with high probability.

Consider all pairs (a, b) of coprime positive integers bounded by

$$L^{0.9509418059\cdots+o(1)}.$$

There are $L^{1.9018836118\cdots+o(1)}$ such pairs. Sieve the integers $a - bm$, using all primes up to $L^{0.9509418059\cdots+o(1)}$, to see which integers are smooth. This takes time $L^{1.9018836118\cdots+o(1)}$ on a general-purpose computer with $L^{0.9509418059\cdots+o(1)}$ bits of memory.

Both a and b are bounded by $L^{o(1)\alpha}$, and m is bounded by $L^{(0.7133923253\cdots+o(1))\alpha}$, so each $a - bm$ is bounded by $L^{(0.7133923253\cdots+o(1))\alpha}$. It is conjectured that the fraction of smooth integers is $\exp(-(1 + o(1))u \log u)$, where

$$u = \frac{(0.7133923253\cdots + o(1))\alpha}{0.9509418059\cdots + o(1)} = (0.7501955649\cdots + o(1))\alpha;$$

this means that there are $L^{1.9018836118\cdots-0.7501955649\ldots/3+o(1)} = L^{1.6518184235\cdots+o(1)}$ pairs (a, b) for which $a - bm$ is smooth.

Now, for each integer k up to $L^{0.1250325942\cdots+o(1)}$, and for each of the

$$L^{1.6518184235\cdots+o(1)}$$

pairs (a, b) where $a - bm$ is smooth, check smoothness of

$$N_k(a, b) = a^d + f_{d-1}a^{d-1}b + \cdots + (f_1 + k)ab^{d-1} + (f_0 - km)b^d,$$

using all primes up to $L^{0.9509418059\cdots-0.1250325942\cdots+o(1)} = L^{0.8259092117\cdots+o(1)}$. The constant in the k bound is $9/(92 + 26\sqrt{13})^{2/3}(-5 + 2\sqrt{13})$.

Coppersmith checks smoothness here with the elliptic-curve method, which takes time $L^{o(1)}$ per integer, totalling

$$L^{1.6518184235\cdots+0.1250325942\cdots+o(1)} = L^{1.7768510177\cdots+o(1)}.$$

See [3] for another method.

It is commonly believed that this use of ECM makes Coppersmith's variant impractical. Standard practice is to instead sieve $N_k(a,b)$ over all (a,b,k). However, there are relatively few pairs (a,b) for which $a-bm$ is smooth; for large n, this outweighs any speed advantages of sieving.

The quantity $N_k(a,b)$ is bounded by

$$L^{(0.7133923253\cdots+0.9509418059\cdots\cdot 1.4017532352\cdots+o(1))\alpha} = L^{(2.0463780783\cdots+o(1))\alpha}.$$

Consequently it is conjectured that there are

$$L^{1.7768510177\cdots-(2.0463780783\ldots/0.8259092117\ldots)/3+o(1)} = L^{0.9509418059\cdots+o(1)}$$

pairs (a,b) for which both $a-bm$ and $N_k(a,b)$ are smooth.

Every such pair is a "relation mod n" among $L^{0.9509418059\cdots+o(1)}$ primes of various number fields. It is conjectured that there will be more relations than primes, if the $o(1)$ in the bound on a and b is chosen large enough, so there will be a nontrivial dependency modulo 2 among those relations. One can discover such a dependency in time $L^{1.9018836118\cdots+o(1)}$ on a general-purpose computer with $L^{0.9509418059\cdots+o(1)}$ bits of memory: apply Wiedemann's algorithm to the relation matrix.

Finally, perform a square-root computation to find a divisor of n. This takes time just $L^{0.9509418059\cdots+o(1)}$ on a general-purpose computer with $L^{0.9509418059\cdots+o(1)}$ bits of memory. The divisor is conjectured to be a nontrivial factor of n with probability bounded away from 0.

Circuits. One can use, instead of a general-purpose computer, the machine described in section 5 to find pairs (a,b) for which $a-bm$ and $N_k(a,b)$ are smooth, and the machine described in section 4 to find a dependency in the relation matrix.

However, since the machine in section 5 is relatively fast, it is better to consider more pairs (a,b), so as to reduce the time spent on linear algebra, when n is sufficiently large. One can balance the time taken by the two machines as follows.

Define α and L as before. Select an integer degree $d \in (1.4227573217\cdots+o(1))\alpha$ with $d \ge 2$, and select $m, f_{d-1}, \ldots, f_0$ as before. The constant here is $(5/3)^{1/3}(6/5)$.

Consider all pairs (a,b) of coprime positive integers bounded by

$$L^{0.9880259179\cdots+o(1)},$$

and select $y \in L^{0.7904207343\cdots+o(1)}$. The constants here are $(5/3)^{1/3}(5/6)$ and $(5/3)^{1/3}(2/3)$.

Find all pairs (a,b) for which $a-bm$ and $N_0(a,b)$ are both y-smooth. This takes time

$$L^{2\cdot 0.9880259179\cdots-0.7904207343\cdots+o(1)} = L^{1.1856311014\cdots+o(1)}$$

on a machine of size $L^{0.7904207343\cdots+o(1)}$, as explained in section 5. The product of $a-bm$ and $N_0(a,b)$ is bounded by

$$L^{2/1.4227573217\cdots+0.9880259179\cdots\cdot 1.4227573217\cdots+o(1)} = L^{2.8114422176+o(1)}$$

so the number of relations is conjectured to be

$$L^{2\cdot 0.9880259179\cdots-(2.8114422176\ldots/0.7904207343\ldots)/3+o(1)} = L^{0.7904207343\cdots+o(1)}$$

which, as before, should exceed the number of relevant primes. Finding a dependency takes time $L^{1.5\cdot 0.7904207343\cdots+o(1)} = L^{1.1856311014\cdots+o(1)}$ on a machine of size $L^{0.7904207343\cdots+o(1)}$, as explained in section 4. The final square root takes time $L^{0.7904207343\cdots+o(1)}$ on a general-purpose computer of size $L^{0.7904207343\cdots+o(1)}$.

Plans. I already have tools that accurately predict the yield of the number field sieve for various parameter choices. It should be straightforward to optimize these choices, given the exact costs of the computations described in sections 4 and 5.

Credits. I started thinking about the *cost* of factorization—rather than simply the time taken on common general-purpose computers—after I heard a talk by Arjen Lenstra on TWINKLE. See [9].

Silverman in [20] pointed out that many previous analyses of the difficulty of factorization were wildly underestimating the cost of sieving and linear algebra. I agree. Silverman's estimates were much more accurate. However, they are now obsolete.

References

[1] —, *Proceedings of the 18th annual ACM symposium on theory of computing*, Association for Computing Machinery, New York, 1986. ISBN 0–89791–193–8.

[2] Alfred V. Aho (chairman), *Conference record of the fifth annual ACM symposium on the theory of computing*, Association for Computing Machinery, New York, 1973.

[3] Daniel J. Bernstein, *How to find small factors of integers*, to appear, Mathematics of Computation. Available from `http://cr.yp.to/papers.html`.

[4] Richard P. Brent, Fred G. Gustavson, David Y. Y. Yun, *Fast solution of Toeplitz systems of equations and computation of Padé approximants*, Journal of Algorithms **1** (1980), 259–295. MR 82d:65033.

[5] Don Coppersmith, *Modifications to the number field sieve*, Journal of Cryptology **6** (1993), 169–180. MR 94h:11111.

[6] Erich Kaltofen, Austin A. Lobo, *Distributed matrix-free solution of large sparse linear systems over finite fields*, Algorithmica **24** (1999), 331–348. MR 2000b:65093.

[7] Donald E. Knuth, *The art of computer programming, volume 3: sorting and searching*, 2nd edition, Addison-Wesley, Reading, 1998. ISBN 0–201–89685–0.

[8] Robert Lambert, *Computational aspects of discrete logarithms*, Ph.D. thesis, 1996. Available from `http://www.cacr.math.uwaterloo.ca/techreports/2000/lambert-thesis.ps`.

[9] Arjen K. Lenstra, Adi Shamir, *Analysis and optimization of the TWINKLE factoring device*, in [16] (2000), 35–52.

[10] Hendrik W. Lenstra, Jr., *Factoring integers with elliptic curves*, Annals of Mathematics **126** (1987), 649–673. MR 89g:11125.

[11] 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. MR 94m:11107.

[12] Hendrik W. Lenstra, Jr., Robert Tijdeman (editors), *Computational methods in number theory I*, Mathematical Centre Tracts 154, Mathematisch Centrum, Amsterdam, 1982. ISBN 90–6196–248–X. MR 84c:10002.

[13] Robert T. Moenck, *Fast computation of GCDs*, in [2] (1973), 142–151.

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

[15] Carl Pomerance, *Analysis and comparison of some integer factoring algorithms*, in [12] (1982), 89–139. MR 84i:10005.

[16] Bart Preneel (editor), *Advances in cryptology: EUROCRYPT 2000*, Lecture Notes in Computer Science 1807, Springer, Berlin, 2000. ISBN 3–540–67517–5.

[17] Manfred Schimmler, *Fast sorting on the instruction systolic array*, Report 8709, Christian Albrecht University Kiel, 1987.

[18] Claus P. Schnorr, Adi Shamir, *An optimal sorting algorithm for mesh-connected computers*, in [1] (1986), 255–261.

[19] Arnold Schönhage, *Schnelle Berechnung von Kettenbruchentwicklugen*, Acta Informatica **1** (1971), 139–144.

[20] Robert D. Silverman, *A cost-based security analysis of symmetric and asymmetric key lengths*, Bulletin 13, RSA Laboratories, Bedford, Massachusetts, 2000. Available from `http://www.rsasecurity.com/rsalabs/bulletins/index.html`.

[21] C. D. Thompson, H. T. Kung, *Sorting on a mesh-connected parallel computer*, Communications of the ACM **20** (1977), 263–271.

[22] Douglas H. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Transactions on Information Theory **32** (1986), 54–62. MR 87g:11166.

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

E-mail address: djb@pobox.com