Factoring Small Integers: An Experimental Comparison

Jérôme Milan

INRIA Futurs, École polytechnique, CNRS
jerome.milan@lix.polytechnique.fr

Abstract. In this data-based paper we report on our experiments in factoring integers from 50 to 200 bits with the postsieving phase of NFS as a potential application. We implemented and compared several general-purpose factoring algorithms suited for these smaller numbers, from Shanks's square form factorization method to the self-initializing quadratic sieve, and revisited the older continued fraction method testing new practical ideas. We give detailed timings for our implementations to better assess their relative range of practical use on current hardware.

1 Introduction

It is almost cliché to recall that integer factorization is a challenging algorithmic problem, be it from a purely theoretical or from a more pragmatic standpoint. On the utilitarian side, the main interest in integer factorization certainly stems from cryptography given the ubiquity of the RSA cryptosystem, based on the premise that factoring large integers is computationally impracticable.

The best general-purpose factoring algorithm currently known, the Number Field Sieve (NFS), factors a number N in $L_N(1/3)$. It relies on the factorization of a lot of much smaller residues for which its recursive application is inefficient.

We focus here on general-purpose factoring algorithms more suitable to factor these small NFS residues (say from 50 to 100 bits) arising in the postsieving phase, and hence known to have no small factors. Fast factorization of medium-sized integers (*e.g.* from 100 to 200 bits) has other practical uses (computing group structures, discrete logarithms, *etc.*) and constitutes our second area of interest.

Motivated by the question "how fast can we factor small integers in the most predictable way" we restrict ourselves to algorithms that can factor any integer in a given time, without assumption on the size or the properties of the factors. Essentially for this reason, we discarded efficient but in some sense "specialized" algorithms such as Pollard's rho and $p-1$, or Lenstra's elliptic curve method (ECM).

In this paper, we give preliminary results for several implementations in the 50 to 200 bit range. We give detailed timings and carefully break down the cost of each algorithm according to their main steps. In particular, we provide precise numbers on CFRAC using either the classical early abort variation or

the newer batch methods for smooth residue detection. We also report on an unusual approach towards QS/SIQS where we tried to use the batch methods in the postsieving phase. Concerning the smaller composites, we contribute to assess the practicable range of SQUFOF.

The next section briefly describes the implemented algorithms. We then give a detailed account of our timing measurements and discuss the implication of these results from a down-to-earth perspective.

2 Implemented algorithms

We selected several general-purpose factoring methods likely to perform well in the range of numbers we are interested in, that is to say from 50 to 200 bits. For the smaller integers in that range, the obvious candidate is Shanks's SQUFOF method. We also implemented an enhancement to Fermat's algorithm described by McKee in [6] as an alternative to SQUFOF. For the numbers in the middle of our range of interest, we revisited the now old-fashion continued fraction algorithm improved with Bernstein's batch method to identify the smooth residues. Finally, we implemented the quadratic sieve, both in it's basic variant and its self-initializing improvement.

The present section merely scratches the surface of the implemented algorithms. Our goal here is not to give an exhaustive theoretical background for all the methods but only to identify their main stages to understand the timing results which will be presented in the third section. Readers unfamiliar with the following algorithms are advised to consult the given references where more detailed descriptions and complexity analyses are available.

2.1 Shanks's SQuare Form Factorization

The SQUare Form Factorization method (SQUFOF) [5], proposed in the mid-seventies by Shanks as an alternative to the continued fraction method, factors a number N in $O(N^{1/4})$. One of its chief advantages is that most of the computations involve numbers less than $2\sqrt{N}$ which makes it particularly suited to factor double precision numbers. However, while relatively easy to implement, SQUFOF's theoretical background is very complex and is out of the scope of this paper so we will merely give the main idea of the method.

SQUFOF is based on the theory of quadratic forms $F(x, y) = ax^2 + bxy + cy^2 \equiv (a, b, c)$ of positive discriminant $\Delta \equiv b^2 - 4ac$. More precisely on the underlying structure of cycles of reduced forms $(a_i, b_i, c_i) = \rho^i(a_0, b_0, c_0)$ where ρ is the standard reduction operator. Finding a point of symmetry in a cycle of forms (a_i, b_i, c_i) of discriminant $\Delta = 4N$ leads to a simple relation giving an hopefully non-trivial factor of N. The theory of quadratic forms is tightly linked to continued fractions and the reduction operator can be expressed in the same formalism as the continued fraction's. Given a number to factor N, we recall the following relations arising from the development of $\sqrt{N}$ in continued fractions:

$$q_0 = \lfloor N \rfloor \qquad , \qquad q_i = \left\lfloor \frac{q_0 + P_i}{Q_i} \right\rfloor \text{ for } i > 0 \tag{1}$$

$$P_0 = 0 \quad , \quad P_1 = q_0 \tag{2}$$
$$P_i = q_{i-1}Q_{i-1} - P_{i-1} \text{ for } i > 1 \tag{3}$$
$$Q_0 = 1 \quad , \quad Q_1 = N - q_0^2 \tag{4}$$
$$Q_i = Q_{i-2} - q_{i-1}(P_{i-1} - P_i) \text{ for } i > 1 \tag{5}$$

Moreover we have the pivotal equality:

$$N = P_m^2 + Q_{m-1}Q_m \tag{6}$$

The principal cycle of reduced forms can then be written as the set of quadratic forms $F_i = \rho^i(F_0) = ((-1)^{(i-1)}Q_{i-1},\ 2P_i,\ (-1)^i Q_i)$ with the principal form $F_0 = (1, 2q_0, q_0^2 - N)$. We will now recall the main steps of the algorithm.

[1 - forward: find square form]
Forward cycle through the cycle starting with the principal form $F_0 = (1, 2q_0, q_0^2 - N)$ until we identify a square form, *i.e.* a form $F_n = \rho^n(F_0) = (-Q, 2P, S^2)$.

[2 - invsqrt: inverse square root]
Take the inverse square root of the square form to get a form $F^{-1/2} = (-S, 2P, SQ)$ and compute its reduction $G_0 = (-S_{-1}, 2R_0, S_0)$ where $S_{-1} = S$, $R_0 = P + S\lfloor (q_0 - P)/S \rfloor$ and $S_0 = (N - R_0^2)/S$.

[3 - reverse: find symmetry point]
Using eqs (1) to (5) replacing Q_i by S_i, P_i by R_i and q_i by t_i, reverse cycle through the forms $G_i = \rho^i(G_0) = ((-1)^{(i-1)}S_{i-1},\ 2R_i,\ (-1)^i S_i)$ to find a symmetry point, in other words a pair of forms $G_m, G_m{+}1$ with $R_m = R_{m+1}$. We know from the theory that this should happen for $m \approx n/2$. Then, using eqs (3, 5) we write $R_m = t_m S_m/2$ and since $N = R_m^2 + S_{m-1}S_m$, we obtain a factorization of N: $N = S_m(S_{m-1} + S_m t_m^2/4)$.

This description skipped a lot of details, and in particular is silent on how to detect a proper square form, *i.e.* a form providing a non trivial factorization. As in other factoring methods, one can use a multiplier k, leading to factor kN instead of N hoping to achieve some kind of speed up. We refer the reader to [5] for a full theoretical background on these issues. Finally, note that in step 3, we have a rough estimate of the number of forms to explore. Thus by using form compositions, one can jump over forms in the cycle to get quickly in the vicinity of the point of symmetry, a strategy dubbed "fast return" by Shanks [10].

2.2 McKee's speedup of Fermat's algorithm

In [6], J. McKee proposed an improvement to Fermat's famous factoring algorithm which heuristically runs in $O(N^{1/4})$ instead of $O(N^{1/2})$. We implemented the so-called "greedy" variant which is presented as being the fastest on a workstation. McKee described this algorithm as a possible alternative to SQUFOF and showed some competitive timings when compared to Shanks's method.

Define $b = \left\lceil \sqrt{N} \right\rceil$ and $Q(x, y) = (x + by)^2 - Ny^2$. As in Fermat's algorithm, we seek x and y such that $Q(x, y)$ is a square so that $\gcd(x + by - \sqrt{Q(x, y)}, N)$ is potentially a proper factor of N. The "greedy" variant proceeds as follows:

[1. Compute modular square root]
Choose a prime p greater than $2N^{1/4}$ and compute the solutions to $Q(x, 1) \equiv 0 \pmod{p^2}$. There is at most 2 solutions which we denote by x_0 and x_1.

[2. Find square]
For $x_i \in [x_0, x_1]$: Set $x = x_i$ and $y = 1$. While $Q(x, y)$ is not a square, set $r = \lceil p^2/x \rceil$, $x = xr - p^2$ and $y = r$. Abort the loop when y exceeds a given threshold y_{max} in the order of $N^{1/4}$.

If no factor is found, go back to step 1 and choose a different prime p. Abort the algorithm when p reaches a chosen bound.

This description assumes that N has no factor less than $2N^{1/4}$ so that the prime p chosen in step 1 should be greater than $2N^{1/4}$. However, as emphasized by McKee, trying with smaller primes may also lead to success and removes the aforementioned "no small factor" assumption.

2.3 The Continued FRACtion algorithm

All the other factoring methods we implemented belong to the "congruence of squares" family of algorithms, so-called because they look for relations of the form $X^2 \equiv Y^2 \pmod{N}$ to factor an integer N. However, they do not attempt to find directly such (X, Y) pairs, focusing instead on the easier task to find several congruences c_i of the form $x_i^2 \equiv y_i \pmod{N}$ at the expense of finding afterwards a sub-collection $\{c_j\}$ of these congruences such that $\prod_j y_j$ is a square. Finding such a sub-collection is achieved by factoring the y_i on a factor base $\mathcal{B} = \{p_1, \ldots, p_l\}$ and solving a linear algebra system over $\mathbb{F}_2$. Since such a sub-collection may only give a trivial factor, $l + \delta$ congruences c_i (with δ small, typically 16) are usually collected to make sure to have multiple solutions. Note that we usually relax the smoothness condition on the y_i to allow them to be the product of a $\mathcal{B}$-smooth number by one of more primes greater than p_l. These are the so-called large prime variations.

The continued fraction algorithm (CFRAC) was introduced in 1975 by Morrison and Brillhart in [7], which is our main reference for our implementation. It runs in $L_N(1/2, \sqrt{2})$ and presents the advantage to generate residues y_i less than $2\sqrt{N}$. Congruences are found by computing the partial quotients q_i of the continued fraction expansion of $\sqrt{N}$ as given in eqs (1) to (5) and the numerator of the i^{th} convergent given by:

$$A_0 = 1 \quad , \quad A_1 = q_0 \quad , \quad A_i = q_i A_{i-1} + A_{i-2} \ \text{ for } i > 1 \tag{7}$$

The congruences sought are then given by the relation:

$$(-1)^i Q_i = A_i^2 \pmod{N} \tag{8}$$

where Q_i is given in eqs (4, 5). The factor base $\mathcal{B}$ is obtained by keeping only the primes p_i such that N (or kN if a multiplier is used) is a quadratic residue mod p_i. The multiplier k, if any, is chosen so that the factor base contains as many small primes as possible. The main steps of CFRAC (without large primes) can be summarized as follows:

[1 - genrel: generate relations]
Expand $\sqrt{N}$ (or $\sqrt{kN}$) as continued fractions to generate $l + \delta$ congruences c_i of the form (8).

[2 - selrel: select relations]
If y_i is not smooth on the factor base, discard the congruence. Otherwise, keep the relation and factor $y_i = \prod_{j=1}^{l} p_j^{e_{ij}}$ on the base, which gives a new row in a $(l + \delta) \times l$ binary matrix M whose coefficients are given by the e_{ij} mod 2. Go back to step 1 to generate new relations until we have filled the matrix.

[3 - linalg: solve linear system]
Solve the linear system given by the decomposition of the y_i to find sub-collections of relations $\{c_j\}$ for which $\prod_j y_j$ is a square, or in other words, find the kernel of the matrix M.

[4 - dedfac: deduce factors]
Since the linear system is surdetermined, we have found δ sub-collections $\{c_{ij}\}_{i=1..\delta}$. For each sub-collection, compute $Y_i^2 = \prod_j y_{ij}$ and $X = \prod_j x_{ij}$. A possible factor of N is given by $\gcd(X - Y, N)$.

The large prime variations only alter step 2 of the algorithm. We refer to [7] for a description of the single large prime variation which is the only one we implemented, albeit in the context of batched smooth number detection (cf 2.5).

2.4 The Quadratic Sieve and the Self-Initializing Quadratic Sieve

The Quadratic Sieve (QS) and Self-Initializing Quadratic Sieve (SIQS) essentially differ from CFRAC in the way the relations c_i are generated. Compared to CFRAC, their main advantage lies in the sieving process which, by discarding most of the non-smooth y_i, lowers their complexity to $L_N(1/2, 3/\sqrt{8})$.

QS was proposed by C. Pomerance in [9] as an enhancement to Schroeppel's linear sieve algorithm. The residues y_i are taken as the values of the quadratic polynomial $g(x) = (x+b)^2 - N$, where $b = \lceil \sqrt{N} \rceil$. The key idea is that if a prime p in the factor base divides $g(x_0)$ (which can only happen if N is a quadratic residue mod p) then p also divides $g(x_0 + ip)$, $i \in \mathbb{Z}$, making possible to use a sieve to tag values of x for which $g(x)$ is smooth. The main step of the algorithm are recalled below:

[1 - polyinit: initialize polynomial]
The factor base is fixed by choosing a set of primes $\mathcal{B} = \{p_0, \ldots, p_l\}$ such that N (or kN when using a multiplier) is a quadratic residue modulo each p_i. Also compute $\{x_{0i}\}$ and $\{x_{1i}\}$ such that $g(x_{0i}) \equiv 0 \pmod{p_i}$ and $g(x_{1i}) \equiv 0 \pmod{p_i}$.

[2 - fill: fill sieve]
A sieve is used to tag x values for which $g(x)$ is divisible by each p_i of the base. One can for example initialize the sieve with zeroes and add $\log p_i$ at the positions $x_{0i} + jp_i$ and $x_{1i} + jp_i$, $j \in \mathbb{Z}$.

[3 - scan: scan sieve and select relations]
Values of x for which $g(x)$ is smooth should have a sieve value of $\log g(x)$. In practice a small correction term is applied since we do not sieve with all prime powers. For the positions x_i in the sieve fulfilling this criteria, compute $g(x_i)$ and check if it is $\mathcal{B}$-smooth. Each smooth $g(x_i)$ gives a new line in an $(l + \delta) \times l$ matrix like in the CFRAC method. Go back to step 2 while the matrix is not filled.

[4 - linalg & 5 - dedfac]
These last two steps are generic to "congruence of squares" methods and are thus similar to CFRAC's.

While an improvement over CFRAC, QS has a serious drawback in that the residues y_i generated are no longer bounded above by $2\sqrt{N}$ but grow as $2x\sqrt{N}$ which lowers the probability to find smooth residues. A suggestion by P. Montgomery known as MPQS (Multiple Polynomial QS) is to sieve with several polynomials $g_{ab}(x) = (ax+b)^2 - N$ discarding one g_{ab} when the residues get too large, which leads to sieve only on an interval $[-M, +M]$. However switching polynomials requires recomputing the solutions to $g_{ab} \equiv 0 \pmod{p_i}$, which can become costly. The Self-Initializing QS (SIQS), basically a variant of MPQS, intends to bypass this problem by choosing a quadratic coefficient $a \simeq \sqrt{2N}/M$ as a product $p_{a0}, \ldots, p_{as}$ of primes not in the base such that the new polynomial obtained by changing the linear coefficient b can be quickly derived from a previous polynomial with the same a. However, only 2^s polynomial g_{ab_i} can be quickly derived from a polynomial g_{ab_0}, after what, a new polynomial with a different a must be fully initialized as in MPQS. We once again voluntarily put aside the details and refer the reader to [4] for a full description.

[1 - polyinit: initialize polynomial]
Polynomials are initialized as briefly sketched above.

[1.1. Full polynomial initialization]
Choose a as a product of primes not in $\mathcal{B}$. The first polynomial g_{ab_0} must be fully initialized as in MPQS.

[1.2. "Fast" polynomial initialization]
If the current polynomial is g_{ab_i} with $i < 2^s$, a new polynomial $g_{ab_{i+1}}$ can be quickly derived from g_{ab_i} otherwise, goto step 1.1.

[2 - fill: fill sieve]
This step is similar to the 'fill' stage in QS, with the exception that we only sieve on an interval $[-M, +M]$. If the interval $[-M, +M]$ has already been completely sieved, go back to step 1 to switch polynomial.

[3 - scan & 4 - linalg & 5 - dedfac]
These last step are performed in the same way as in the basic QS.

2.5 Identifying smooth residues

Testing the smoothness of a lot of residues on the factor base is a major bottleneck of the "congruence of squares" methods. While the use of the sieve greatly mitigates its extent in the case of QS or SIQS, it remains by far the most time-consuming stage of the CFRAC algorithm.

Identifying the smooth residues has traditionally been achieved by trial-division. The early abort variation can significantly reduce the cost of this phase by beginning to trial-divide with only a fraction of the primes in the base and discarding the partially factored residues if they exceed a given bound. But, however interesting it may be in practice, the need of a better way to detect smooth numbers remains.

In [1], D. Bernstein introduced a simple but efficient batch algorithm to find the $\mathcal{B}$-smooth parts of a large set of integers $\{x_i\}$. His method, based on calculus trees, is shown to run in $O\big(b \cdot \log^2(b) \cdot \log(\log(b))\big)$ where b is the total number of bits in $\mathcal{B}$ and $\{x_i\}$. We reproduce below almost verbatim the algorithm described in [1] (which we will dub "smoothness batch") as a convenience for the reader.

[Input] A factor base $\mathcal{B} = \{p_1, \ldots, p_l\}$ and a set of integers $S = \{x_1, \ldots, x_m\}$.

[Output] The set of integers $S' = \{x'_i | x'_i$ is the $\mathcal{B}$-smooth parts of $x_i\}$.

[1 - prodtree: compute primes product]
Compute $z = p_1 \times \cdots \times p_l$ using a product tree.

[2 - remtree: product modulo the $\{x_i\}$]
Compute $Z' = \{z'_i | z'_i = z \bmod x_i\}$ using a remainder tree.

[3 - powm: compute modular powers]
Compute $Y = \{y_i | y_i = {z'_i}^{2^e} \bmod x_i\}$ where $e > 0$ is the smallest integer such that $2^{2^e} \geq x_i$.

[4 - gcd: compute smooth parts]
Compute $S' = \{x'_i | x'_i = \gcd(x_i, y_i)\}$.

Note that this algorithm does not give the factorization of the x_i on the base. However for our application, this phase (which we will refer to as 'factres' in the plots of section 3) is generally negligible and can be carried out via simple trial division without much impact on the total running time. As a result of step 4, if $y_i = 0$ then x_i is $\mathcal{B}$-smooth, otherwise the cofactor x_i/x'_i may be prime and hence considered as a large prime.

3 Implementation details and results

Our programs were developed and benchmarked on an AMD Opteron 250 workstation with 2GB of RAM running under the GNU/Linux operating system. They are single-threaded and aimed to be run on a single core/processor. Multi-precision computations were performed using the GMP library version 4.2 patched with Pierrick Gaudry's assembly routines for Opteron processors. The programs,

written exclusively in C99 for portability reason, were compiled with the `gcc` compiler version 4.0 using the optimization flags `'-march=opteron -O3'`.

We want to point out that the presented timings are still preliminary results as our programs still have significant room for improvement. We do not claim to have the fastest implementations of the selected algorithms. For example, our SIQS implementation is still slower than Pari's MPQS. However, given that our implementations all share a large part of identical underlying code, it is our opinion that the comparisons drawn between the different methods are valid.

All composites used in our experiments are the product of two primes of comparable sizes chosen at random. Factorization timing for one composite was obtained by cumulating and averaging over the CPU time taken by n_e factorizations, where n_e varies from a few hundreds to 1 depending on the size of the composite.

Relation selection and smoothness batch We selected smooth residues using Bernstein's batch algorithm adapted to take into account the large primes. Product and remainder trees were programmed in the standard fashion, using the low level MPN functions of the GMP library.

Note that in our experiments we took the unusual approach to also use batches to select relations for QS and SIQS, even though the use of a sieve theoretically makes such batches redundant. The basic idea was to relax the sieving criteria and to collect more residues hoping to find more relations via the large prime variation and to cut the time needed to fill the sieve. Obviously such a strategy can only make sense if one has a fast smoothness test.

Linear algebra For the size of the numbers we are interested in, the linear algebra phase does not significantly impact the total running time. Consequently, we settled for gaussian elimination following the algorithm given in [8]. For the largest factor bases involved (about 6000 primes), the linear phase typically takes less than 5% of the total running time.

SQUFOF Our implementation follows closely the continued fraction description given in [5]. In order to use mostly single precision computations, our program is restraint to factor numbers fitting in two machine words, or 128 bits on now widespread 64 bits processors. Note that this limitation is not a cause of concern since SQUFOF is only useful to factor very small integers.

Factorization is first attempted without using a multiplier as our experiments show that this is noticeably faster. If this first attempt fails to find a factor, we perform a sequential race using the 16 square free multipliers suggested in [5], which is virtually guaranteed to succeed.

To carry out the "fast return", we used the "large step" algorithm described by Williams and Wunderlich in [12] in the context of the parallel generation of residues for CFRAC. This is the only significant part of the program that requires multi-precision computations.

Fig. 1 a) shows the mean timings for SQUFOF in a range from 45 to 80 bits. 1000 composites were factored for each size shown. The 'reverse' step of our program only performed the "fast return" variation if the estimated number of forms to scan was greater than a given bound n_{FR}, to take into account the slower multi-precision computations involved and the fact that the form cycle should then be scanned in both directions. Our implementation uses $n_{FR} = 4096$ which has been determined experimentally. Fig. 1 a) also shows the improvement obtained by using a "fast return" compared to a standard reverse cycle. Unsurprisingly, the speed-up is found to be much less that the purely theoretical value of 33% since the 'reverse' step of the algorithm is already much faster as shown in Fig. 1 b). One of the bottleneck of the 'forward' step is the square detection. For our implementation (based on a slightly improved variant of algorithm 1.7.3 given in [3]) the square detection still represents more than 15% of the total running time for a 70-bit composite.

We observed no failure during our experiments since we perform a multiplier race if needed (in about 5% to 10% of the time for the composite sizes considered). While we took the 16 multipliers from [5], it is likely that using less multipliers lead to a slight speed-up with a minimal impact on the failure rate.

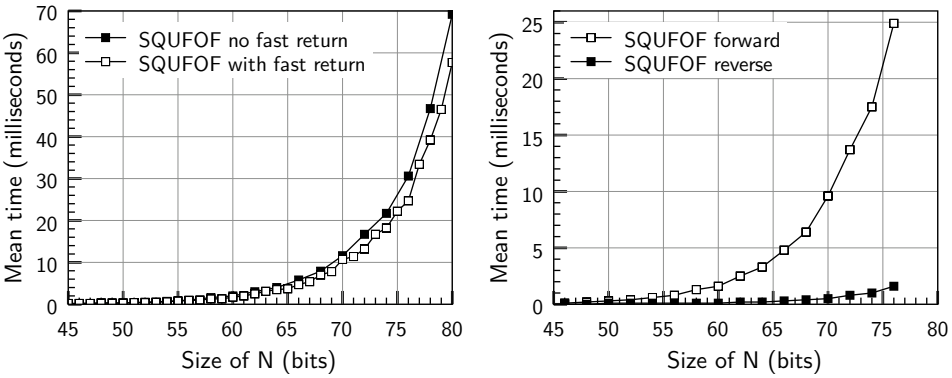

Fig. 1: Left: SQUFOF timings from 45 to 80 bits (1000 composites per size) with and without fast return. Right: Relative impact of the 'forward' and 'reverse' steps for SQUFOF with fast return.

McKee's Fermat variation We implemented the "greedy" variant of McKee's algorithm as given in [6]. This variation was implemented for essentially two reasons: it is presented as being faster on a workstation and McKee provides timings for it which makes for an easy comparison. We took $y_{max} = N^{1/4}/10$ and began with the smallest prime greater than $N^{1/4}$ exactly as in McKee's paper. We stop the factorization after having used one million different primes, a large bound that is never reached in practice for the small numbers we considered. Our program uses single precision functions whenever possible and is restricted to double precision integers at most.

It is suggested in [6] to perform a race with the multipliers 1 and 3. In Fig. 2, we provide timings for a race with the multiplier lists $\{1, 3\}$ and $\{1, 3, 5, 7, 11\}$

for tiny 45 to 60 bit composites. We did not observed any failure in that size range.

Our measurements are in disagreement with McKee's in that SQUFOF appears significantly faster, at least with our implementations. We also tried to begin with primes much smaller than $2N^{1/4}$ but failed to achieve any significant speed-up. While we do not discard the possibility for our "greedy" Fermat implementation to be particularly slow, two others reasons could account for such a discrepancy. First, timings given in [6] are restricted to a tiny sample of composites which may not be statistically significant. Second, we found Maple's SQUFOF implementation used in [6] as a reference, to be incomparably slower that ours on the same computer.

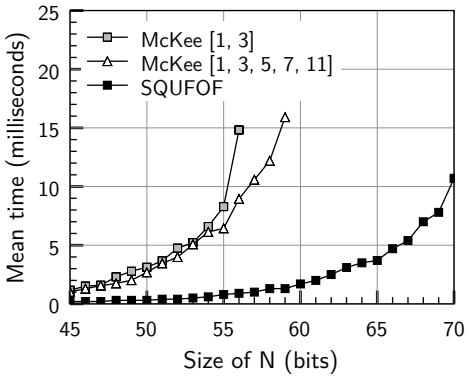

Fig. 2: McKee's "greedy" Fermat timings using the multipliers {1, 3} and {1, 3, 5, 7, 11} averaged over 100 composites per size, compared to SQUFOF's.

CFRAC Apart from the choice of base size, our implementation of CFRAC is straightforwardly adapted from the description given by Morrison and Brillhart in [7] and uses the single large prime variation. Selection of the smooth residues is achieved either via trial-division (with or without early abort) or using the aforementioned "smoothness batch" algorithm followed by a trial-division step. Batches are performed with only 128 residues at a time as it seemed faster. Early abort is programmed as suggested in [9]: if p_k is the greatest prime in the base, we first begin to trial-divide with primes less that $\sqrt{p_k}$ and discard any partially factored residues other a given bound chosen empirically.

Our timings obtained for CFRAC are presented in Fig. 3. We selected the $\mathcal{B}$-smooth residues according to three methods: naive trial division, trial division with (one step) early abort and smoothness batches. The sizes of bases given in table 1 were determined for each method independently. We notice that using batches makes possible to use much larger bases.

In [9], Pomerance's asymptotic analysis gives N^{1-c}, with $c = 1/7$, as the optimal bound to use in the early abort variant. However we found this theoretical value to be of little practical use. After experimenting with some multiples of c's "optimal" value, we settled for $c = 4/7$ which seemed to work best for us, although we caution that this could certainly be fine-tuned.

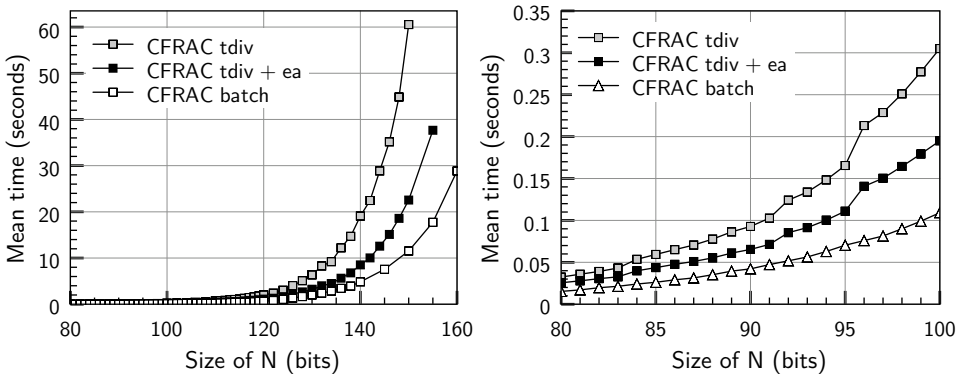

Fig. 3: Left: CFRAC timings obtained with smooth selection of the residues by trial division only (tdiv), by trial division and early abort (tdiv+ea) and by smoothness batch (batch). Right: Zoom on the 80-100 bits range.

CFRAC tdiv (+ ea)

Size of N (bits)	Size factor base
80	128
95	160
105	192
110	256
130	384
140	448
145	512

CFRAC batch

Size of N (bits)	Size factor bases
60	128
92	256
110	512
125	1024
145	2048
172	4096

Table 1: Left: Sizes of the factor bases used for CFRAC with trial division (with or without early abort). Right: Sizes of the bases used for CFRAC with batches. The given values are restricted to a selected set of composite sizes.

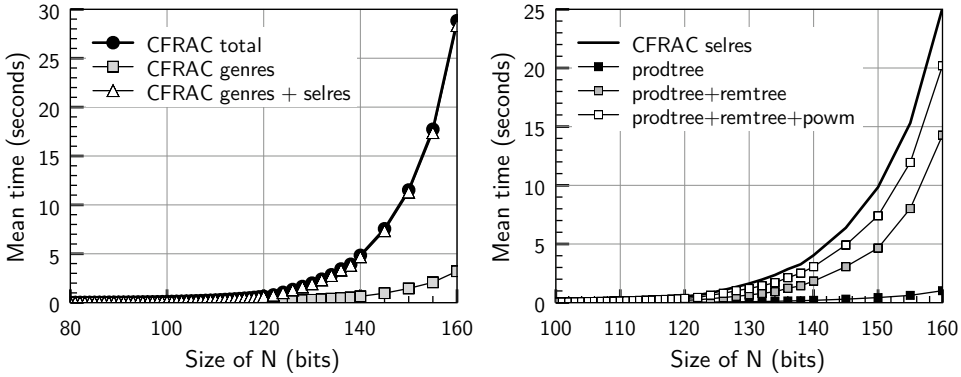

Fig. 4: Left: Timings for the two first steps of the CFRAC algorithm with batches (generate relations and select relations) compared to the total running times. Right: Timings for the different steps of the smoothness batches.

Fig. 4 a) shows the relative impact of the different steps involved in CFRAC. Despite the use of smoothness batches, the relation selection still dominates

the running time by a fair margin. Note that for such small bases, the actual factorization of the residues on the base is negligible just like the linear algebra phase even if the naive gaussian elimination is used.

Fig. 4 b) shows the relative impact of each steps of the smoothness batches. The real bottleneck comes from the computations of the remainder trees and, to a much lesser extent, the modular powers at each node. However, our remainder trees are obtained from a semi-naive implementation. It would certainly be interesting to implement Bernstein's scaled remainder trees described in [2] as these could alter quite significantly CFRAC's total running time.

QS and SIQS Our main references for the implementation of QS and SIQS are [11,4]. We chose a multiplier $k < 100$ such that $kN \equiv 1 \pmod 8$ and to maximize the number of small primes for which kN is a quadratic residue using Silverman's modified Knuth-Schroeppel function [11].

SIQS polynomial selection is still extremely crude: the quadratic coefficients a are obtained by multiplying primes above a given bound until $\sqrt{2N}/M$ is reached. To reduce the amount of redundant relations, we followed Contini's recommendation from [4] and made sure that the quadratic coefficients' decompositions always differ by at least two primes.

We do not sieve with the 10 smallest prime from the base. Residues are computed from the values of the sieve S such that $S[x] \geq \kappa \cdot \log(g(x)/p_{next})$ where p_{next} is the first prime greater than the largest prime in the base and κ is a correction factor what we took equals to 0.7.

Fig. 5 shows the running time of QS and SIQS for the empirical selection of parameters given in table 2. We do not claim that these values are optimal for a given composite size (much less for a given number) but they gave the best average results out of the numerous parameter combinations we tried, although there is certainly room for improvements.

Details of the relative timings for each steps of the algorithm are given in Fig. 6. Filling and scanning the sieve accounts for the major part of the running time, typically between 75% and 90% of the total time.

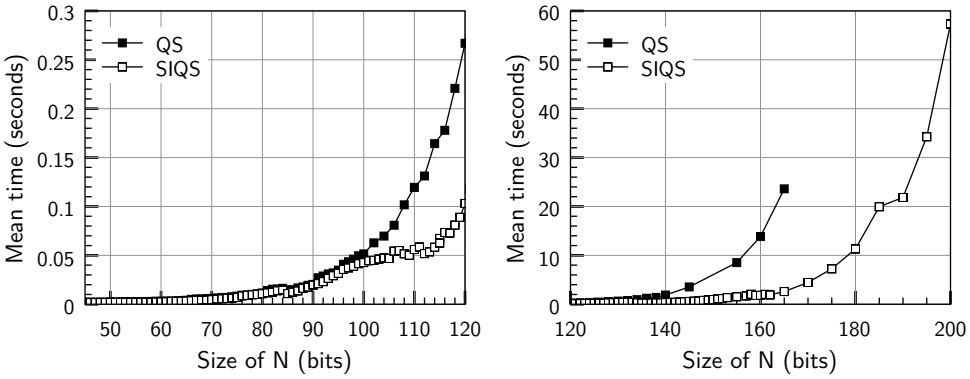

Fig. 5: QS and SIQS timings over the whole 45-200 bit range. Each data point is obtained from an average over 100 composites.

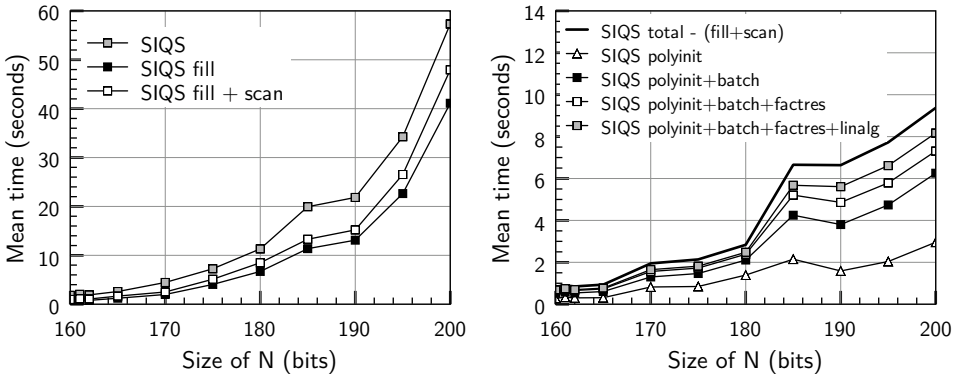

Fig. 6: Timings for each step of the SIQS algorithm. The left figure shows the two most time-consuming stages. The right figure details the contributions of the remaining phases.

QS	
Size of N (bits)	Size of base
45	92
100	512
125	1024
145	1536
155	3072
165	5120

SIQS		
Size of N (bits)	Size of base	M
45	64	5,000
100	384	15,000
125	512	25,000
150	1024	25,000
170	3072	75,000
185	5120	100,000
200	7168	250,000

Table 2: Left: Size of the bases used for QS. Right: Size of the bases and sieve half-width used for SIQS. The given values are restricted to a selected set of composite sizes.

Overall comparison Fig. 7 gives an overall comparison of the implemented algorithms our range of interest. SQUFOF is found to remain competitive up to 64 bits beyond what SIQS, while not optimally implemented, is the clear winner.

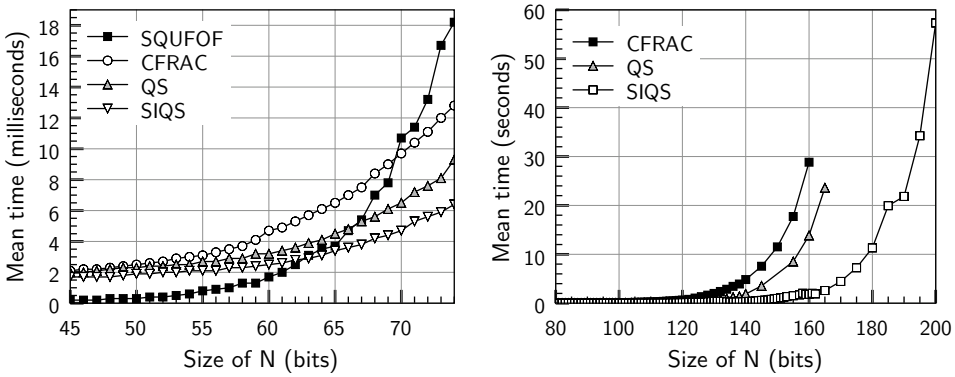

Fig. 7: Comparison of the implemented algorithms over the 45-200 bit range.

4 Conclusion

The comparisons provided in this paper are part of an on-going work to precisely assess the current practicality of factoring methods with an emphasis on small to medium-sized composites. Our timings are still preliminary and are expected to improve in the short term. Our experiments suggest that SQUFOF is indeed unbeatable for factoring the tiniest integers, up to 64 bits. However, we remain skeptical about the practical use of McKee's improvement to Fermat's method. We provided real-world numbers showing that smoothness batches have a significant impact on CFRAC's running time. While CFRAC is still quickly superseded by the sieve methods, it would certainly be worthwhile to investigate the impact of a more efficient implementation of the remainder trees on its running time. Surprisingly, SIQS appears to be the most practical method for numbers as small as 64 bits. However, our implementation remains largely perfectible and more work is needed to carefully assess the leeway that batches bring to sieve methods, if any. Finally, our implementations are part of a larger software package that will eventually be released under an open source license and made available at http://www.lix.polytechnique.fr/Labo/Jerome.Milan/tifa/tifa.html.

References

1. D. J. Bernstein. How to find smooth parts of integers, 2004. http://cr.yp.to/papers/sf.pdf.
2. D. J. Bernstein. Scaled remainder trees, 2004. http://cr.yp.to/arith/scaledmod-20040820.pdf.
3. H. Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, NY, 1st edition, 1993.
4. S. Contini. Factoring integers with the self-initializing quadratic sieve, 1997. http://citeseer.ist.psu.edu/contini97factoring.html.
5. J. E. Gower and S. S. Wagstaff Jr. Square form factorization. *Mathematics of Computation*, May 2007.
6. J. McKee. Speeding Fermat's factoring method. *Mathematics of Computation*, 68(228):1729–1737, October 1999.
7. M. A. Morrison and J. Brillhart. A method of factoring and the factorization of F_7. *Mathematics of Computation*, 29(129):183–205, January 1975.
8. D. Parkinson and M. C. Wunderlich. A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers. *Parallel Computing*, 1:65–73, 1984.
9. C. Pomerance. Analysis and comparison of some integer factoring algorithms. In H. W. Lenstra, Jr. and R. Tijdeman, editors, *Computational Methods in Number Theory, Part I*, pages 89–139. Mathematisch Centrum, Amsterdam, 1982.
10. D. Shanks. Unpublished notes on SQUFOF, circa 1975 (?). http://cadigweb.ew.usna.edu/~wdj/mcmath/shanks_squfof.pdf.
11. R. D. Silverman. The Multiple Polynomial Quadratic Sieve. *Mathematics of Computation*, 48(177):329–339, January 1987.
12. H. C. Williams and M. C. Wunderlich. On the Parallel Generation of the Residues for the Continued Fraction Factoring Algorithm. *Mathematics of Computation*, 48(177):405–423, January 1987.