Integer Factorization and Computing Discrete Logarithms in Maple

Aaron Bradord[*], Michael Monagan[*], Colin Percival[*]
Department of Mathematics, Simon Fraser University,
8888 University Drive,
Burnaby, BC, Canada V5A 1S6.
anbradfo@sfu.ca, mmonagan@cecm.sfu.ca, cperciva@irmacs.sfu.ca

1 Introduction

As part of our MITACS research project at Simon Fraser University, we have investigated algorithms for integer factorization and computing discrete logarithms. We have implemented a quadratic sieve algorithm for integer factorization in Maple to replace Maple's implementation of the Morrison-Brillhart continued fraction algorithm which was done by Gaston Gonnet in the early 1980's. We have also implemented an indexed calculus algorithm for discrete logarithms in GF(q) to replace Maple's implementation of Shanks' baby-step giant-step algorithm, also done by Gaston Gonnet in the early 1980's.

In this paper we describe the algorithms and our optimizations made to them. We give some details of our Maple implementations and present some initial timings. Since Maple is an interpreted language, see [7], there is room for improvement of both implementations by coding critical parts of the algorithms in C. For example, one of the bottle-necks of the indexed calculus algorithm is finding and integers which are B-smooth. Let B be a set of primes. A positive integer y is said to be B-smooth if its prime divisors are all in B. Typically B might be the first 200 primes and y might be a 50 bit integer.

2 Integer Factorization

Starting from some very simple instructions — "make integer factorization faster in Maple" — we have implemented the Quadratic Sieve factoring algorithm in a combination of Maple and C (which is accessed via Maple's capabilities for external linking). In this section, we shall describe some the design decisions we made, the reasoning behind them, and some of the optimizations we were able to make which allowed us to improve performance.

[*] This work was supported by the MITACS NCE of Canada.

2.1 Maple

The first design decisions we made resulted from the target of speeding up integer factorization *in Maple*. While most work in the field of integer factorization has historically been centered around factoring integers which are as large as possible — usually involving hundreds of processors and several months — Maple is typically used interactively and on a single processor, so we concluded that obtaining good performance for inputs ranging from 50 to 100 digits (which, on modern systems take time ranging from a few CPU-seconds to a couple of CPU-days) was of far greater importance than reducing the time needed to factor a 150-digit input. In light of this, we decided to ignore the General Number Field Sieve in favour of the Quadratic Sieve. While the Quadratic Sieve is asymptotically slower than the Number Field Sieve, the constants involved make the Quadratic Sieve faster up to a break-even point of around 90–135 digits depending upon the implementations used, and even beyond this point the Quadratic Sieve falls behind quite slowly.

Our second major design decision was inspired by the use of Maple as a pedagogical and research tool. The Quadratic Sieve factoring algorithm consists of three major steps — sieving to find "relations"; filtering the list of relations found to make it more manageable; and solving a large sparse linear system modulo 2 — and all three of these steps are targets of active research. Consequently, we decided to perform as much work as possible in Maple, and to return the results of each step into Maple before passing it into the next step. In this manner, we make it easier for students to examine the code in order to learn how the Quadratic Sieve operates, and even more importantly make it easier for researchers to "drop in" a replacement for one of the parts if they are investigating potential improvements. Although Maple is generally significantly slower than code written directly in C, we shall see later that since most time is spent executing a relatively small amount of code, it is possible to obtain good performance while only writing a small portion of the code in C.

2.2 The Sieve

While it is traditional to speak of *the* Quadratic Sieve, it is grammatically more accurate to speak of *a* Quadratic Sieve, since there have been a series of improvements made to the original concept. The original Quadratic Sieve developed by Pomerance [10] in 1981, which factored an input N by searching for values of x such that $f(x) = (x + \left\lfloor \sqrt{N} \right\rfloor)^2 - N$ is smooth, was quickly replaced by the Multiple Polynomial Quadratic Sieve [11] (hereafter MPQS), which takes a series of values $a = q^2$ for primes q, solves $b^2 \equiv N \pmod{a}$, and then searches for smooth values of $f(x) = (ax + b)^2 - N$. This was revised further by Alford and Pomerance [1] in the Self-Initializing Quadratic Sieve[1] (hereafter SIQS) which takes values a of the form $a = q_0 q_1 \ldots q_r$ for appropriate

[1] The Self-Initializing Quadratic Sieve was also independently discovered by Peralta, who called it the Hypercube Multiple Polynomial Quadratic Sieve.

q_i and then searches for smooth values of $f(x) = (ax + b_i)^2 - N$ for b_i equal to the 2^r non-trivially different[2] square roots of N modulo a.

Whichever Quadratic Sieve is being used, the sieving takes a similar form. An array of M bytes is first initialized to zero, then for each prime p less than a bound B_1 (known as the "prime bound") modulo which N is a quadratic residue, each position x in the array satisfying $f(x) \equiv 0 \pmod{p}$ is increased by $\log p$. These locations are computed from the roots of $f(x)$ modulo p — for each root α, the values in positions $\alpha, \alpha + p, \alpha + 2p, \ldots$ are adjusted. In SIQS, this initialization cost — computing the roots of $f(x)$ modulo each of the sieving primes — is amortized over 2^r sieving intervals (thus the "Self-Initializing" sieve would more appropriately be called an "Amortized Initialization Costs" sieve).

In light of our design decision to perform as much computation as possible within Maple, and also in keeping with the general principle of using the best available algorithms, we decided to use SIQS, performing the initialization computations in Maple but sieving each "hypercube" in C.

We also apply the ubiquitous "large prime variation", with either one or two large primes, depending upon the size of integer to be factored: Rather than requiring that $f(x)$ factor completely over the primes less than B_1, we permit it to have one or two prime factors between B_1 and a "large prime bound" B_2.

2.3 Choosing hypercubes

The choice of the primes q_i (and thus the values a) is subject to a number of considerations:

1. In SIQS, as in MPQS, the value a should be approximately equal to $\sqrt{8N}/M$, where M is the length of the sieving interval[3]. The further a varies from this value, the larger the average value of $f(x)$ will be at points within the sieve, with the inevitable result that fewer of the points within the sieve will be smooth.

2. Since SIQS amortizes the cost of sieve initialization over 2^r sieving intervals, larger values of r will reduce the time spent on initialization.

3. On the other hand, each prime q has the effect of reducing the frequency of smooth values in the sieve. Since N is required to be a quadratic residue modulo each q, there would normally be two values $x \bmod q$ for which $x^2 - N \equiv 0 \pmod{q}$; however, for $b^2 \equiv N \pmod{q}$ there will only be one value of $x \bmod q$ for which $f(x)q^{-1} \equiv 0 \pmod{q}$.

4. Finally, SIQS can often find the same relation multiple times, which both decreases the number of useful relations found and adds the additional complexity of removing such duplicates. If a value $0 \leq x < \sqrt{2N}$ satisfies $x^2 - N \equiv 0$ modulo both $a = q_0 \ldots q_r$ and $a' = q'_0 \ldots q'_r$, then the relation

[2] Since $(-ax - b)^2 = (ax + b)^2$, we can ignore half of the 2^{r+1} square roots of N modulo a; this is usually performed by requiring that all the b_i are congruent modulo q_0.

[3] Note that some authors use a sieving interval of length $2M$, i.e., from $-M$ to M.

will be found twice (assuming that both values are taken for a during the sieving). Naturally, this is almost impossible unless the sets $\{q_0 \ldots q_r\}$ and $\{q_0' \ldots q_r'\}$ intersect.

While Contini [5] recommends minimizing the number of duplicate relations found by requiring that the sets $\{q_i\}$ differ by at least two primes, we adopt a sharply different approach. Fixing $q_1 \ldots q_r$, we allow q_0 to take on, in increasing order, the primes immediately following the sieving prime bound B_1. Since the number of hypercubes which need to be sieved in order to find enough relations is typically far less than B_1, the values a will remain reasonably close to their ideal value and the number of duplicate relations will be small. In addition, since all of the values q_0 are above the prime bound B_1, any duplicate relation will contain the relevant q_0' as a "large prime", making the task of removing duplicates trivial.

Choosing the primes q_i in this manner has a further benefit in reducing the initialization costs. After fixing $q_1 \ldots q_r$, we can precompute

$$\begin{aligned}
\alpha_0 &= \prod_{i=1}^{r} q_i \\
\beta_i &= \alpha_0 q_i^{-1} && \forall 1 \le i \le r \\
\gamma_i &= \sqrt{N} \beta_i^{-1} \bmod q_i && \forall 1 \le i \le r \\
\theta_{i,p} &= q_i^{-1} \bmod p && \forall 1 \le i \le r, \forall p \in P \\
\psi_p &= \sqrt{N} \alpha_0^{-1} \bmod p && \forall p \in P
\end{aligned}$$

where P is the set of primes less than B_1 modulo which N is a quadratic residue, and the notation $\sqrt{N} \bmod p$ means either root of $x^2 - N \bmod p$.

For each cube, after selecting q_0, we then compute

$$\begin{aligned}
\alpha_i &= q_0 \beta_i && \forall 1 \le i \le r \\
\phi_i &= \gamma_i q_0^{-1} \bmod q_i && \forall 1 \le i \le r \\
\phi_0 &= \sqrt{N} \alpha_0^{-1} \bmod q_0 \\
\theta_{0,p} &= q_0^{-1} \bmod p && \forall p \in P
\end{aligned}$$

and note that $a = q_0 \beta_0$, the 2^r values of b are

$$-\alpha_0 \theta_0 - \sum_{i=1}^{r} d_i \alpha_i \phi_i$$

where the d_i each take the values -1 and 1, and the sieving offsets at which $\log p$ must be added are

$$(\phi_0 \pm \psi_0)\, \theta_{0,p} + \sum_{i=1}^{r} d_i \theta_{i,p} \phi_i .$$

In this manner, the initialization costs are reduced r-fold beyond those in most implementations of SIQS.

2.4 Sieving optimizations

The first optimization we made to the sieving process itself was to apply the widely used “small prime variation”. Here we fix a value B_0, known as the “small prime bound”, and instead of sieving using primes less than B_0, we compute the average contribution which will result from such primes. Since small primes require the most work to be done during sieving, this provides a significant speedup. In our implementation, however, we differ somewhat from the usual small prime variation: Instead of merely computing the mean contribution from small primes, we compute both the mean and standard deviation, and then attribute to each point in the sieve an estimated small prime contribution equal to the mean plus 2.5 standard deviations[4]. In this manner, we reduce the number of smooth values which are missed due to having above-average numbers of small divisors.

After the sieving process produces a list of “hits”, we add an optimization which we believe to be new. Realizing that most points will have fewer small prime divisors than the above-average value which we earlier attributed to them, we compute for each point the *actual* contribution from small primes. Taking this in combination with the “over-sieve” (the amount by which the sieve threshold was exceeded), we filter the list of hits returned by the sieving and reduce the number of insufficiently smooth points being considered; this provides a significant speedup in later steps.

Given this (now much reduced) list of sieve hits, we proceed to trial division: We divide the values $f(x)$ by each of the sieving primes p in turn, constructing the factorization of each $f(x)$ into a product of primes less than B_1 and possibly a remaining term which is either a single large prime or the product of two large primes. In this trial division process, we make another new optimization: Since we know from the sieving process the sum of the (rounded) logarithms of the primes dividing $f(x)$, we subtract away the logarithm of each prime as we find it, and exit the trial division loop once we know that we have found all the sieving prime divisors. Since the largest prime divisor of $f(x)$ below B_1 is on average about $\frac{2}{3}B_1$, this “early exit” optimization provides a significant speedup to the trial factorization step.

Once the trial factoring is complete, we consider the remaining part (if any) of $f(x)$. If the remaining value is between B_1 and q_0, we discard the value — we have a duplicate of a relation which was found in an earlier hypercube. For values between q_0 and B_2, we have a relation containing a single large prime. Values between B_2 and B_1^2 we discard, since they must reflect a single prime greater than the large prime bound; and finally values greater than B_1^2 are subjected to a single pseudoprime test, probable primes are discarded, and then Brent’s variation of the Pollard Rho factoring algorithm [3] is used to split the value and a relation is constructed containing the two large primes.

[4] The value 2.5 was chosen based on performance testing.

2.5 Cycle counting and relation filtering

In keeping with standard practice, we use the Union-Find algorithm to find cycles within the large primes. Unlike most implementors, however, we have two major advantages: First, since all the sieving is performed on the same system, we can run the Union-Find algorithm while the sieving is ongoing, adding new relations as they are found; this allows us to stop sieving as soon as we have enough relations rather than needing to guess when we should stop. Second, Maple provides very easy to use hash tables, making the implementation far simpler than it would be in a less powerful language.

Once we have enough relations, we take further advantage of the Union-Find algorithm to filter the relations. By effectively running the algorithm backwards, we eliminate all of the relations which contain a large prime but are not a member of a cycle; this is considerably faster than the more common approach of repeatedly filtering the list of relations by removing "singletons" (relations containing a large prime which does not occur in any other relations).

Finally, we make a pass through the relations eliminating any prime which occurs in only two relations by multiplying those relations together.

2.6 Linear algebra

Once we have a filtered list of relations with more relations than primes involved, we return to C and apply the block Lanczos algorithm for solving sparse systems of linear equations modulo 2. Here we do not diverge from the usual approach at all: For performance reasons, the entire block Lanczos algorithm must be performed in C[5], and we use a block size equal to the machine word-length, taking advantage of the internal parallelism of bitwise operators.

Once the block Lanczos algorithm completes, we take the solutions returned in order, multiply together the indicated relations to produce an equation of the form $X^2 \equiv Y^2 \pmod{N}$, and compute `igcd(`$X - Y$`, `N`)`; when we find a non-trivial factor, we stop and return it to the user.

2.7 Performance

In order to demonstrated the performance of our code, we consider the values

$$N = p_>(10^{n/2}e) \cdot p_>(10^{n/2-1}\pi)$$

where $p_>(x)$ denotes the smallest prime greater than x; these are n-digit integers. In Table 1 we show the time taken on a 2.5GHz Apple G5 system by Maple's current default integer factoring algorithm, the Morrison-Brillhart algorithm ("maple"), Magma's integer factorization routine, which spends a short time searching for small factors using special-purpose factoring algorithms prior

[5] We found at one point that our C code could solve an entire system of order 2000 in less time than it took for Maple to compute a single matrix-vector product!

to using the Multiple Polynomial Quadratic Sieve ("magma"), our own implementation ("SIQS"), and Jason Papadopoulos' "msieve" command-line application, which is generally regarded as being the fastest implementation available of any form of the Quadratic Sieve [9].

n	maple	magma	SIQS	msieve	n	magma	SIQS	msieve
28	0.72	0.26	0.19	0.15	60	87.73	21.66	14.59
32	2.41	0.38	0.32	0.19	64	204.49	49.90	42.14
36	5.89	0.49	0.47	0.26	68	449.85	92.27	81.39
40	19.60	1.15	0.81	0.51	72	1189.31	226.69	201.47
44	84.83	2.04	1.91	0.63	76	2731.23	425.95	396.17
48	281.29	4.87	2.93	1.33	80	7177.24	1329.67	1329.88
52	1472.33	11.89	5.73	3.18	84	$> 10^4$	3339.39	2821.30
56	5136.75	32.34	10.58	6.21	88	$> 10^4$	7027.39	5042.18

Table 1: Integer factorization timings (in CPU seconds)

While it is immediately clear that the Morrison-Brillhart algorithm is inferior to the various Quadratic Sieve implementations, we note also that the gap between Magma's MPQS implementation and our SIQS reaches a factor of 5, while our code comes very close to msieve's performance, confirming that — at least for large inputs — the cost of performing hypercube initialization in Maple is ultimately insignificant.

3 Discrete Logarithms

We begin by defining the discrete logarithm problem. Suppose we have a finite group G and an element $\alpha \in G$ of order n and an element $\beta \in \langle\alpha\rangle$. The Discrete Logarithm Problem (DLP) is, given α and β, solve $\alpha^c = \beta$ for the unique integer c satisfying $0 \leq c < n$. We denote this value, c, as $\log_\alpha(\beta)$.

Example 1. Let G be the subgroup of $\mathbb{Z}_{101}$, the integers modulo 101, generated by $\alpha = 2$ and suppose $\beta = 14$. Here $G = \langle\alpha\rangle = \{1, 4, 16, 64, 54, 14, 56, ...\}$ and the order of α is 50. In this simple example, the order of α is small, so we may simply test if $\alpha^c = \beta$ for c from 0 to 50 until we find one that works, in this case, $c = 5$.

In a larger group, for example, a group with 2^{100} elements, this brute force method, which requires $n/2$ multiplications in G on average, becomes infeasible. Although we can do better than $O(n)$ multiplications in general, it turns out that there are no known algorithms for large subgroups of $\mathbb{Z}_q$ generated by α which are polynomial time in $\log q$. It is this difficulty of computing discrete logarithms in this group, and also the group of points on an elliptic curve, that is the basis for various modern cryptographic systems, such as the the ElGamal public key cryptosystem. See [12].

3.1 Maple's `mlog` command.

Maple's current method, the `numtheory[mlog]` command, for solving the DLP on the multiplicative structure of $\mathbb{Z}_m$ where m is an integer, employs the following strategy. The method factors m and solves a number of DLPs in $\mathbb{Z}_p$ for each prime p dividing m. To solve these smaller problems, Maple implements the Pohlig-Hellman algorithm, which in turn calls Shanks' baby-step, giant-step algorithm as a subroutine. The Pohlig-Hellman algorithm takes advantage of the factorization of

$$(p-1) = \Pi_{i=1}^{k} q_i^{e_i}$$

by solving e_i instances of the DLP modulo q_i to obtain c_i satisfying

$$\beta = \alpha^{c_i} \bmod q_i^{e_i},$$

and then applies Chinese remainder theorem to find the value c modulo $(p-1)$. For a full description of this algorithm, we refer the reader to the *Handbook of Applied Cryptography* [6] or Stinson's text [12].

To solve the DLPs modulo each prime factor q of $(p-1)$, Maple currently implements Shanks' Baby-Step, Giant-Step method. This method is is an $O(\sqrt{q})$ method. It is illustrated with the following example in $\mathbb{Z}_{101}$.

Example 2. Suppose $q = 101$, $\alpha = 7$ and $\beta = 57$. Here α is a primitive element of $\mathbb{Z}_{101}$ so $n = 100$. In the first step we build a table of pairs of values. Let $k = \lceil \sqrt{q} \rceil = 11$ and for i from 1 to $k-1$ compute values $(i, \alpha^{mi} \bmod q)$ and sort them by the second entry. In our example, this table is

i	0	7	4	9	6	3	1	8	5	10	2
α^{ki}	1	15	19	29	30	38	51	58	69	65	76

In the second step we compute the discrete logarithm of β as follows. For j from 0 to $k-1$ we test, using binary search, if $\beta \times \alpha^{-j}$ is among the second entries in the table. For $j = 0$ we have $\beta\alpha^{-j} = 57$, which is not in the table. Eventually we find for $j = 8$ we have $\beta\alpha^{-j} = 15$, and we see that (7,15) is one of the entries in the table. We have $\alpha^{11\times 7} \equiv \beta\alpha^{-8} \pmod{q}$ and so $\beta \equiv \alpha^{11\times 7+8} \pmod{q}$ and we have found that $\log_\alpha \beta = 11 \times 7 + 8 = 85$.

Now let us look at the complexity of the algorithm. Notice that we can compute the k entries α^{mi} in the table in $O(k) = O(\sqrt{q})$ multiplications in G. The cost of sorting the table on the second entry using an $O(n \log n)$ sorting algorithm requires $O(k \log k) = O(\sqrt{q} \log q)$ comparisons. In the second step, computing the $\beta\alpha^{-j}$ costs one inverse and at most another $k = O(\sqrt{q})$ multiplications and at most another $O(k \log k) = O(\sqrt{q} \log q)$ comparisons to search the table using binary search. Thus this algorithm does $O(k) = O(\sqrt{q})$ multiplications in G and makes $O(k \log k) = O(\sqrt{q} \log q)$ comparisons of elements of G. It also needs to store $O(\sqrt{q})$ elements of G in the table.

This is not a problem for Maple's current implementation when the prime factors of $(p-1)$ are all small, however, when this is not the case, Maple will take an inordinate amount of resources to compute the desired logarithm. In the worst case, p is prime and $p = 2q+1$ with q also prime. This will force Maple to construct a table with $O(\sqrt{p})$ entries.

3.2 The Index Calculus Method

As we will see, a better approach would be to use an index calculus method instead of Shanks' method when the prime q is 'large'. The index calculus method is a randomized algorithm that constructs a database of the logarithms of 'small' primes (say the smallest 50, 100 or 500 primes) and uses this database to reconstruct the logarithm desired. Returning to our previous example, we illustrate how the index calculus method works.

Example 3. Recall that we had $q = 101$, $\alpha = 7$ and $\beta = 57$ where α is of order $n = 100$. Let $B = \{2, 3, 5\}$ be the set of 'small' primes (called the factor base). We calculate $\log_\alpha 2$, $\log_\alpha 3$, and $\log_\alpha 5$ and use these to reconstruct $\log_\alpha \beta$. The key is to find values of c such that $\alpha^c \bmod q$ is B-smooth (that is, is only divisible by the primes in B, or, in an abuse of notation, is divisible only by primes less than or equal to B; in this way, B is an upper bound but can also be thought of as a set of primes). We choose these values of c at random from $[2, q-2]$. Many values of c will work, three of which are $c = 25, 30, 61$. For these values we have

$$\alpha^{25} = 2 \times 5 \bmod q, \;\; \alpha^{30} = 2 \times 3 \bmod q, \;\; \alpha^{61} = 2 \times 5^2 \bmod q.$$

Taking logarithms this yields the system of linear congruences

$$\begin{aligned} 25 &= \log_\alpha 2 + \log_\alpha 5 \pmod{q-1}, \\ 30 &= \log_\alpha 2 + \log_\alpha 3 \pmod{q-1}, \\ 61 &= \log_\alpha 2 + 2\log_\alpha 5 \pmod{q-1}. \end{aligned}$$

One now solves this linear system modulo $q - 1 = 100$ to obtain the unique solution which is

$$\{\log_\alpha 2 = 89, \;\; \log_\alpha 3 = 41, \;\; \log_\alpha 5 = 36\}.$$

Now, by finding one value d for which $\beta\alpha^d$ is B-smooth, we can solve for $\log_\alpha \beta$. One value that works is $d = 81$. We obtain

$$\beta\alpha^{81} = 2 \times 3 \times 5 \pmod{q}.$$

Again, taking logarithms we have

$$\log_\alpha \beta + 81 = \log_\alpha 2 + \log_\alpha 3 + \log_\alpha 5 \pmod{q-1}.$$

Substituting for $\log_\alpha 2$, $\log_\alpha 3$ and $\log_\alpha 5$, and solving for $\log_\alpha \beta$ we obtain

$$\log_\alpha \beta = 89 + 41 + 36 - 81 = 85 \pmod{q-1}.$$

3.3 Optimizing the index calculus algorithm.

The limiting factor for this procedure is how quickly we can find a set of congruences that determines a unique solution for the logarithms of the factor base B. For large q, few choices of c work, so we generally end up spending most of the time searching for appropriate candidates c. In order to make this task faster, we might simply increase the size of the factor base B. While this would certainly guarantee an increase in the probability that a number $\alpha^c \bmod q$ is B-smooth, we also must keep in mind that in order to uniquely determine a system of linear congruences, $\log_\alpha p$ must be involved in at least one congruence for each p in B. But the larger that we make B, the less likely it is that the larger primes of B will be divisors of $\alpha^c \bmod q$. At one extreme $B = \{2\}$ and we expect to search $O(\log_2 q)$ values of c to find $\alpha^c \bmod q$ which is an exact power of 2. This is no better than Shanks' method. And at the other extreme, B contains all primes less than q and we expect to search $O(q)$ values of c to find the only such value to yield a congruence involving the first prime less than q. A compromise can be found between these two extremes (a lot closer to the first one) that will allow us to make significant gains on Shanks' method for large q.

In a result from [4], Canfield et al. show that the probability that a number less than q is B-smooth is asymptotically $u^{-u(1+o(1))}$ where $u = \log_q B$. As well, the prime number theorem tells us that the number of primes less than B is asymptotically $B/\ln B$. Although our choice for B will usually be rather small, these two equations will at least give us a starting point for choosing what value we ought to set B for a given q. We must find at least one congruence for each p in B, and it will take $u^{u(1+o(1))}$ tries to find c for which $\alpha^c \bmod q$ is B-smooth. Therefore, we expect to search $u^{u(1+o(1))}B/\ln B$ choices of c before we have determined our system of congruences. We should, then, seek to minimise this function over all possible choices of B. The above discussion applies in the limit that q and B tend to infinity, however, we will be concerned with 'medium' sized values for q and B such as $\log_q < 100$ and $|B| < 5000$. In practice, we compiled many experiments for different choices of q and B and sought a polynomial approximation relating $\log_2 q$ and B.

Once we have chosen our smoothness bound, B, the greatest speed up that we can get is in formulating a strategy that will allow us to quickly decide whether a value c will lead us to a value for which $\alpha^c \bmod q$ is B-smooth. The greatest boon in this was a result presented in [2] for computing discrete logarithms over $GF(2^k)$, though it is straightforward to modify for logarithms in $GF(q)$ as was done in [8]. We describe the idea.

Consider applying the Extended Euclidean Algorithm (EEA) to the integers q and $y = \alpha^c \bmod q$. We will obtain a sequence of triples (r_i, s_i, t_i) satisfying

$$r_i = s_i y + t_i q$$

where the remainders r_i decrease in size from q to 1 and the s_i increase monotonically in magnitude from 0 to an integer of size $O(q)$. The index, k, of i when r_i is first less than $|s_i|$ will have r_k and s_k roughly the same length, namely, half the length of q. Taking the above equation modulo q we have

$$r_i \equiv s_i \alpha^c \pmod{q}$$

and hence

$$\log_\alpha r_i = \log_\alpha s_i + c \pmod{q-1}.$$

Now instead of checking if $y = \alpha^c \bmod q$ is B-smooth, we check if r_k and s_k are simultaneously B-smooth. It turns out that it is much more likely that these two smaller integers are simultaneously B-smooth than y is. Moreover, we will usually only have to check one of r_k and s_k, because if, say, r_k is not B-smooth, then we don't care about this (r_k, s_k) pair. Furthermore, the pairs

$$(r_{k-2}, s_{k-2}), (r_{k-1}, s_{k-1}), \quad \text{and} \quad (r_{k+1}, s_{k+1})$$

obtained from the Euclidean algorithm will also have integers close to half the length of q in size.

Note also that the sequence s_i alternate from positive to negative, and so one must add the integer -1 to the factor base, B. As well, we don't need to continue the EEA past $i = k + 1$ as we don't make use of any of those values. Lastly, note that we don't need to explicitly compute the t_i values.

3.4 Maple Implementation and Timings

The efficiency of the algorithm now depends on how fast we can test if an integer y is B-smooth. The basic test to see if r_i (or s_i) is B-smooth is to trial-divide by the primes in B in a loop. However, since large primes are far less likely to divide the values r_i than small primes, we handle these differently. For small primes in B we use a traditional trial division loop. For the larger primes in B we perform a gcd calculation with a product of a certain number of them with whatever is left of r_i. If this gcd is 1 then we can skip a bunch of trial divisions. Through experimentation, we found that taking products of twenty large primes at a time and computing the gcd with r_i produced the best gains.

Lastly, a word on the storage of the system of linear congruences. The congruences will be sparse, having typically $O(\log r_i)$ non-zero terms, thus we store the congruences as polynomials rather than in matrix form. Taking advantage of the sparsity of the system will also allow us to make gains in calculating its solution.

With all of these tricks to speed up our Maple implementation of the index calculus algorithm, we find that two thirds of the time is spent on the Euclidean algorithm step for calculating the (r_i, s_i) pairs. This suggests that there is not much else we can do to speed up our Maple implementation unless we adopt another algorithm or code parts of it in C. Below we show timings (in seconds on a 2.0 GHz AMD 64 bit Opteron) which pit Maple's current `mlog` command against our implementation of the index calculus algorithm for increasing choices of a prime q having a large prime dividing $q - 1$ (in fact all choices were with $q = 2p + 1$ with p also prime).

$\lceil \log_2 q \rceil$	Maple's `mlog`	Index Calculus
20	0.009	0.17
25	0.038	0.314
30	0.324	0.587
35	2.744	1.632
40	21.539	2.97
45	164.565	4.572
50	1815.272	8.603
55	27978.411	18.345
60		45.756
65		114.607
70		303.185
75		606.303
80		1875.977
85		5278.757
90		9690.946
95		22124.051

Table 2: Discrete logarithm timings (in CPU seconds)

References

[1] W. Alford and C. Pomerance. *Implementing the Self-Initializing Quadratic Sieve on a distributed network* in A.J. van der Poorten, I. Shparlinski, and H.G. Zimmer (eds), "Number theoretic and algebraic methods in computer science", pp. 163–174, 1995.

[2] I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone. *Computing logarithms in finite fields of characteristic two.* SIAM J. Alg. Disc. Methods, **5**, pp. 276–285, 1984.

[3] R.P. Brent. *An improved Monte Carlo factorization algorithm.* BIT **20**, pp. 176–184, 1980.

[4] E. R. Canfield, P. Erdos, and C. Pomerance. *On a problem of Oppenheim concerning 'factorisatio numerorum.'* J. Number Theory, **17**, pp. 1–28, 1983.

[5] S.P. Contini. *Factoring integers with the Self-Initializing Quadratic Sieve.* MA thesis, University of Georgia, 1997.

[6] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1996. ISBN 0-8493-8523-7

[7] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, P. DeMarco. *Maple 9 Introductory Programming Guide*, Waterloo Maple, 2003. ISBN: 1-894511-43-3.

[8] A. M. Odlyzko. *Discrete Logarithms: The past and the future.* Designs, Codes, and Cryptography, **19**, pp. 129–145, 2000.

[9] J. Papadopoulos. *msieve*, http://www.boo.net/ jasonp/qs.html.

[10] C. Pomerance. *The Quadratic Sieve factoring algorithm.* Proc. EUROCRYPT '84, LNCS 209, 169–182, 1985.

[11] R.D. Silverman. *The Multiple Polynomial Quadratic Sieve.* Math. Comp. **48**, pp. 329–339, 1987.

[12] D. R. Stinson. *Cryptography: Theory and Practice*, CRC Press, 1995. ISBN 0-8493-8521-0