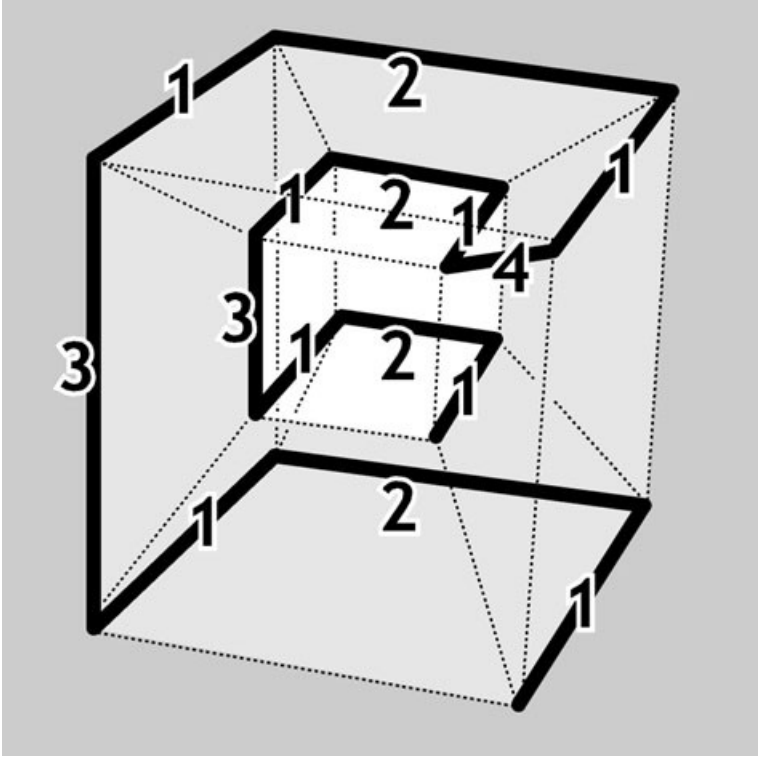
Uli Meyer

Notes on ulimyhmpqs



View on a hypercube of four dimensions.
The Hamiltonian Path reaches each of the 16 corners.

Abstract: HMPQS (Hypercube Polynomial Quadratic Sieve) or SIQS (Self Initializing Quadratic Sieve) is probably the fastest known algorithm for factoring numbers up to approximately 110 decimal digits (for greater numbers the significantly more complex GNFS (general number field sieve) is faster, see [3]). This description refers to ulimyhmpqs, my Common-LISP implementation of the "HMPQS with one large prime". Identifiers related to the source code are high-lighted in this Font.

The intention of the author was not to present a record-breaking implementation of this well-known algorithm, but

- *to understand himself what science of the last decades has in stock, to reverse the very simple and fundamental operation of multiplying two integers, in spite of having only basic knowledge in number theory*
- *to have a ready-to-go, easy-to-use and still powerful LISP module for factoring large numbers.*

We are looking for a non-trivial divisor of $n \in \mathbb{N}$.

If n is odd and composite, as already Fermat knew, it can be expressed as the difference of two squares: $n=(u+v)\cdot(u-v)=u^2-v^2$. As a matter of fact it is sufficient to find two squares, for which this weaker congruence holds:

$$u^2-v^2\equiv 0 \ (mod\ n) \ ,$$

because then $gcd(u+v,n) \mid n$ and $gcd(u-v,n) \mid n$. One can use for instance the Euklidian algorithm to compute the greatest common divisor.

If we had $u\equiv\pm v \ (mod\ n)$ we would find only trivial divisors, and so we look for squares $u^2\equiv v^2 \ (mod\ n)$ with $u\neq\pm v \ (mod\ n)$ and there is a good chance to find non-trivial divisors (the probability depends on the number of divisors of *n*).

QS

The Q*uadratic Sieve* (QS) is a factoring algorithm that searches solutions for $u^2\equiv v^2 (mod\ n)$. It systematically generates many x_i^2-n . The sieve part extracts those that can be multiplied together for a solution. If we found for instance

$$x_1^2-n=p_1\cdot p_2 \ , \ x_2^2-n=p_2^5\cdot p_3 \ \text{ und } \ x_3^2-n=p_3\cdot p_1$$

for three primes $p_{1,}p_{2,}p_3$, it immediately follows that

$$(x_1^2-n)\cdot(x_2^2-n)\cdot(x_3^2-n)=p_1 p_2\cdot p_2^5 p_3\cdot p_3 p_1$$

and we have a solution $(x_1 x_2 x_3)^2\equiv(p_1 p_2^3 p_3)^2 \ (mod\ n)$.

Def. *B-smooth*. Let *B* a subset of all primes. A number is defined to be *B-smooth*, if all its prime factors are contained by *B*. (Note: This extends the common definition for *"x k-smooth"*, which means that no prime factor of *x* is greater than *k*. In this case the set *B* contains all primes less than or equal to *k*.)

In the above example we had a "factor base" $B=\{p_{1,}p_{2,}p_3\}$ and x_1^2-n, x_2^2-n, and x_3^2-n were *B*-smooth. Obviously the QS requires a suitable factor base *B* and on one hand an efficient procedure to find *B*-smooth x_i^2-n (sieving) and on the other hand a method to combine a square from these (solving).

The factor base

Let p be a prime factor of x^2-n, so $p|x^2-n$, and $x^2\equiv n\ (mod\ p)$. As x does exist, n must be quadratic residue modulo p. So the factor base B needs to contain only those primes p, for which the Legendre-symbol is $\left(\frac{n}{p}\right)=+1$. The probability for this is approximately $\frac{1}{2}$, because there are $\lceil\frac{p+1}{2}\rceil$ quadratic residues modulo p. (The Legendre-symbol has been implemented as the generalized jacobisymbol).

It is easier to find B-smooth numbers, if the elements of the factor base are rather small. So we choose $B=\{-1\}\cup\{p\mid p\ prim\wedge p\leqslant p_{max}\wedge\left(\frac{n}{p}\right)=+1\}$.

The -1 is useful, because we extend the definition of *B-smooth* to all integers. Besides always $2\in B$, because $0=0^2$ and $1=1^2$. (Note: It may happen, that the resulting factor base for a certain n lacks many small primes. This reduces the probability, that some x^2-n is B-smooth. Multiplying n now by a small odd prime might result in a more favourable factor base and a faster factorization, although the number to be factored becomes greater. See density.)

Of what size shall the factor base be? The larger B, the easier it is to find B-smooth numbers, but the more B-smooth numbers one needs and the more work it is, to multiply a square from them. Some very rough assumptions about the distribution of smooth numbers and the number of steps needed to find them, lead to the famous result $\#B_{opt}\approx e^{\sqrt{2}/4\cdot\sqrt{\ln n\cdot\ln\ln n}}$ (see [1] and [4] for details). However, the actual optimum value (factor-base-size) is implementation dependent and though limited by the available memory. At this point we are only interested in its magnitude:

For $n\approx 10^{30}, 10^{50}, 10^{70}$ we take $\#B\approx 400, 4\ 000, 25\ 000$ factors.

Sieving

The smaller $|x_i^2-n|$ is, the more likely x_i^2-n is B-smooth. So we search close to $x\approx\sqrt{n}$. For example we could scan all x in an interval $X:=[\lfloor\sqrt{n}\rfloor-M;\lfloor\sqrt{n}\rfloor+M]$ by probe-dividing with the elements of the factor base. That's too slow, and the most important trick of the QS is a quick heuristic to find the x, that have a good chance to give B-smooth x^2-n :

At first the sieve S will be initialized for all $x\in X$ with $S_x\leftarrow\lfloor\log_2|x^2-n|\rfloor$. So we have very small integers here, the function ilog2 is realized by the LISP-function integer-length. Instead of expensive divisions, we simply subtract rounded logarithms. With a little number theory we can find for each $p_j\in B$ all $x\in X$ for which p_j divides x^2-n ...

Assume we know one number r_j with $p_j \mid r_j^2 - n$. Then for all $z \in \mathbb{Z}$, p_j divides $(r_j + z \cdot p_j)^2 - n = r_j^2 - n + (2zr_j + z^2 p_j) \cdot p_j$ as well.

For all $x \in X \cap \{r_j + z \cdot p_j \mid z \in \mathbb{Z}\}$ we now know that $p_j \mid x^2 - n$ and at these $\lfloor (2M+1)/p_j \rfloor$ positions we set $S_x \leftarrow S_x - \lfloor \log_2 p_j \rfloor$. This can be done quite effectively, it's just a loop and some integer additions! After having "sieved in" all $p_j \in B \setminus \{-1\}$ this way, it is easy, to find good candidates for *B*-smooth $x^2 - n$ because at those positions we will have $S_x \approx 0$. In practice we scan the sieve for positions x at which S_x is less than a certain threshold value, for which $\lfloor \log_2 max\, B \rfloor$ is a good heuristic. To find out whether these $x^2 - n$ indeed are *B*-smooth, we then need to probe-divide.

On one hand the smaller factors p_j take the most work to sieve in, on the other hand these small factors have the least effect on S_x . Practice shows, that the run-time can be improved, if primes p_j below a certain limit (omit-below) are not sieved in at all, and to choose a little higher threshold value to compensate. Finally we could also sieve in powers of p_j , but again the computational costs won't pay.

But how do we find r_j , so that $r_j^2 \equiv n \,(mod\, p_j)$? r_j does exist, because *n* is quadratic residue modulo p_j . Instead of simply trying $0, 1, \ldots, p_j - 1$ we use the powerful *Shanks-Tonelli-algorithm*. Its idea is the stepwise "improvement" of an assumed root and it has been implemented as sqrtmodprime.

In fact there always exist two *modular square roots* of *n* for each odd prime p_j : r_j (found by the Shanks-Tonelli-algorithm) and $-r_j \equiv p_j - r_j \,(mod\, p_j)$. It is important to sieve in for both roots.

Solving

How do we proceed, so that a product of *B*-smooth numbers becomes square? Obviously the prime exponents of the product need to become even. The equations we obtained by sieving are of this form: $x_i^2 - n = \prod_{p_j \in B} p_j^{e_{i,j}}$. We are interested in the corresponding congruences $x_i^2 \equiv \prod_{p_j \in B} p_j^{e_{i,j}} (mod\, n)$. By multiplying the congruences for some x_1 and x_2 , we get

$$(x_1 x_2)^2 \equiv \prod_{p_j \in B} p_j^{e_{1,j} + e_{2,j}} (mod\, n) \quad .$$

To find out, if a prime exponent $e_{1,j} + e_{2,j}$ is even, it is sufficient to know if $e_{1,j}$ is even and if $e_{2,j}$ is even, in other words we look at $e_{i,j} mod\, 2$ only and multiplying two congruences becomes an addition of two vectors in $\mathbb{Z}_2^{\#B}$ on the right hand side of the congruence. That sounds like loss of information, but the right hand side can be reconstructed later on (the values of x_1 and x_2 are stored in xarray).

Finally we obtain a matrix that can be solved for instance by the Gauss-algorithm, and at this point we anticipate that it requires approximately $\#B$ equations, to make all prime exponents become even.

Let me put it another way: We transform each equation $x_i^2 - n = \prod_{p_j \in B} p_j^{e_{i,j}}$ into a *relation* of this form:

$$\langle \{i\} \; ; \{ j \mid e_{i,j} \bmod 2 = 1 \} \rangle$$

So we remember only the "non-square" prime factors. The multiplication of two equations is equivalent to the following operation on the two corresponding relations:

$$\langle I_1 ; J_1 \rangle \times \langle I_2 ; J_2 \rangle = \langle I_1 * I_2 \; ; \; J_1 * J_2 \rangle \quad ,$$

where $M_1 * M_2 := (M_1 \cup M_2) \setminus (M_1 \cap M_2)$ means the exclusive-or-operation on the sets M_1 and M_2 (isomorph to addition in $\mathbb{Z}_2^{\#B}$).

The simple algorithm put-into-its-bucket is derived from Gaussian elimination: Each $p_j \in B$ corresponds to an initially empty bucket $bucket_j \leftarrow nil$. Every new relation will be stored in "its" bucket, according to the greatest prime factor that has an odd exponent. If there is already an relation in this bucket, the new relation will be multiplied by the existing one, which makes the corresponding prime factor exponent even and the process is repeated with the resulting relation.

```
put-into-its-bucket (⟨I ; J⟩) :
        if J = {}  then try-to-solve (I)
        else if bucket_maxJ = nil  then  bucket_maxJ ← ⟨I ; J⟩
        else put-into-its-bucket (bucket_maxJ × ⟨I ; J⟩)
```

At latest when all buckets are non-empty, one more relation will cause all exponents to become even and we obtain the relation $\langle I ; \{\} \rangle$, that corresponds to the desired congruence $u^2 \equiv v^2 \pmod{n}$ with

$$u = \prod_{i \in I} x_i \quad \text{and} \quad v = \sqrt{\prod_{i \in I} (x_i^2 - n)} \in \mathbb{N}$$

Note, that at this place we deal with very big numbers, because $\#I \approx \#\frac{B}{2}$ numbers of size $x_i \approx \sqrt{n}$ and $x_i^2 - n \approx M\sqrt{n}$ will be multiplied. Should $gcd(u \pm v, n)$ now obtain only trivial divisors of *n*, every extra relation would give an extra chance.

The ulimyhmpqs-implementation continues the hunt for divisors of *n* until it is completely factored. On new divisors (see new-factor) a probabilistic primality test (prime?, currently Solovay-Strassen) will be applied. Composite divisors are tested for being a power (power-breaker), and tried to be factored further by taking the gcd with already known divisors.

Relations in ulimyhmpqs are not represented as bit-vectors in $\mathbb{Z}_2^{\#B}$ but as sorted

lists. Bit-vectors for new relations would contain almost just 0's (the resulting matrix is called to be *sparse*), corresponding to very short lists. During probe-division the factors occur already in sorted order and the algorithm described above will be supported by the fact, that $max\, J$ must not be searched, as it is the first element of the sorted list. The exclusive-or-operation on two sorted lists (xor-merge) can be done quite effectively by the use of destructive list operations.

In spite of list representation Gaussian reduction is not the optimum method for solving sparse matrices. See [1] for a discussion of this matter, and why more serious implementations use the superior block Lanczos method to delay the point at which the linear algebra step becomes the dominant step in run-time.

Partial relations

If we choose the sieve threshold value too low, we might not catch all possible relations. On the other hand, if we choose the threshold value too high, we might waste time by probe-dividing many numbers that are not *B*-smooth. The first implementations of the QS already used the following trick: If after probe dividing x_1^2-n there is only a "small" remaining factor $R \notin B$,

$$x_1^2-n = R\cdot\prod_{p_j\in B} p_j^{e_{1,j}} \quad ,$$

we keep this "partial relation" in memory. If we now find another partial relation with the same remaining factor *R*, we can multiply both relations and obtain

$$(x_1 x_2)^2 \equiv R^2\cdot\prod_{p_j\in B} p_j^{e_{1,j}+e_{2,j}} \ (mod\ n) \quad .$$

We can use this product as a "full relation", because R^2 is square and needs not to be considered in the solving step. If we have *k* partial relations with *R*, we will get $k-1$ full relations.

This extension is called *QS with one large prime*. In fact it doesn't matter whether *R* is prime, on the other side we know it is for sure if $R<(max\, B)^2$, because *B* is defined to contain all primes below $max\, B$, that possibly divide some x^2-n .

ulimyhmpqs keeps partial relations with *R* less than slp-max (slp means *single large prime*) in a hash-table of 2^h places. A cheap $\lfloor R/2 \rfloor mod\ 2^h$ serves as hash-function. In case of collisions that do not reveal a full relation, only the partial relation with the smaller *R* is kept, as it has the better chance to find a fitting partner. slp-max can be altered by using the alfa parameter (slp-max $\leftarrow$ alfa $max\, B$), *h* corresponds to the slp-log2 parameter, that defaults to 16, and should be increased when factoring numbers greater than 10^{60} if enough memory is available.

The greater *n* is, the more frequently relations will be found by combining partials. For very large numbers $n>10^{100}$ one will find only very few full relations and the algorithm depends almost exclusively on finding partials. Some implementations consequently go one step further, and allow larger remaining factors *R*, that are composite and hence have to be factored. The *QS with two large primes* keeps "partial-partials": If there are e.g. partial relations with $R_1=q_1 q_2$, $R_2=q_2 q_3$, $R_3=q_3 q_1$, a graph-cycle-finding algorithm reveals the desired square

$$R_1 R_2 R_3=(q_1 q_2 q_3)^2$$

See [4] for an excellent discussion of this issue. The authors of [2] estimate that this improvement of the QS has shifted the cross-over-point with the GNFS to approximately 130 decimals.

The size of the sieve

Complying with the general heuristic "start searching, where you believe you will find something", we already mentioned the sieve interval

$$X:=[\lfloor\sqrt{n}\rfloor-M\,;\lfloor\sqrt{n}\rfloor+M\,]$$

above. What size must *M* have, in order to find a few more than $\#B$ relations? This in fact is a difficult question, and can be answered practically by breaking *X* into smaller sub-intervals of size *2m*

$$X_k:=[\lfloor\sqrt{n}\rfloor+(-1)^k\cdot\lfloor k/2\rfloor\cdot m\;\;;\;\;\lfloor\sqrt{n}\rfloor+(-1)^k\cdot\lfloor k/2+1\rfloor\cdot m-1\,]$$

which can be scanned successively for $k=0, 1, 2, ...$ with increasing distance from $[\sqrt{n}]$ until there are enough relations to factor *n*. One has then sieved less than one sub-interval too much, to solve the task (Other than ulimyhmpqs, most implementations first collect the relations before the solving starts. Then of course one needs a probabilistic estimate for the number of required relations, e.g. #*B*+20).

With increasing k, the x^2-n become larger and larger and fewer and fewer of them are *B*-smooth. The QS tends to "fall asleep". Is there a possibility to make the x^2-n smaller and or to limit their size?

MPQS

The QS searches for $x \in X$ with *B*-smooth $f(x)$, where $f(x)=x^2-n$. The principle $p_j | f(s_j) \Rightarrow p_j | f(s_j + z \cdot p_j)$ for all $z \in \mathbb{Z}$ holds though for every polynomial with integer coefficients (not taking into consideration what difficulties might occur, when looking for the suitable s_j).

Consider the polynomial $g(x)=(ax+b)^2-n$. It generates squares modulo *n* as required for the QS. If we now choose *b* so that a given *a* divides b^2-n , then *a* also divides $g(x)=a^2x^2+2abx+b^2-n$ for all $x \in \mathbb{Z}$. The resulting polynomial $h(x) := g(x)/a$ though grows faster than $f(x)=x^2-n$, but by choosing different pairs a , b we can generate as many different polynomials as we like. Instead of enlarging the sieve interval, with the disadvantage described above, one can simply switch to another polynomial. That's the idea of the *Multi Polynomial Quadratic Sieve*, *MPQS*.

Let's have another look on $g(x)$. Let $a \ll n$ and $b \ll n$. The polynomial $g(x)$ gives a quite slim parabola with a minimum of roughly $g(0)=b^2-n \approx -n$. Though we have the best chance to find smooth numbers if we keep them small, and so we choose *a* so that the maximum $g(M)=(a \cdot M+b)^2-n \approx +n$ in the sieve interval $[-M ; +M]$ has approximately the same absolute size. So we obtain

$$a_{ideal} \approx \frac{\sqrt{2n}}{M} \quad .$$

Where the QS sieves on numbers of size $f([\sqrt{n}]+M) \approx 2 \cdot M \cdot \sqrt{n}$, the MPQS with $a \approx a_{ideal}$ comes along with $h(M) \approx n/a \approx \frac{\sqrt{2}}{2} \cdot M \cdot \sqrt{n}$. The sieve size *M* of the QS strictly depends on #*B*. When using the same factor base size with the MPQS, we can greatly influence and limit the size of the numbers sieved upon by simply choosing an *M* much smaller than what would result in the QS. Of course there is an optimum *M*, which varies with different implementations. This optimum depends on the run time costs for switching polynomials and the size of *n*.

The first version of the MPQS used $a \notin B$. To avoid extra considerations in the solving process one chooses $a=t^2$ to be square of some prime *t*. This turned out to be disadvantageous, because the effort to compute all the s_j for each new $a \approx a_{ideal}$ is quite large. Contrarily the HMPQS (or SIQS) uses a that are *B*-smooth and so take part in the solving. Switching polynomials is much faster here, because each *a* turns out to allow a large set of suitable *b*, each giving a different polynomial $g_b(x)$, and the recalculation of the "zeros" s_j within this set of polynomials is relatively cheap.

Modular square roots

The QS required solutions for r_j in $r_j^2 \equiv n \,(mod\, p_j)$. In this context we mentioned the Shanks-Tonelli-algorithm to find modular square roots modulo an odd prime. There are two solutions, $\pm r_j$.

Due to the condition $a \mid b^2 - n$ the MPQS requires to solve another quadratic congruence $b^2 \equiv n \,(mod\, a)$ for each *a*. Having chosen $a = t^2$ as the square of some prime also reveals two solutions for *b*.

Let now $a = \prod a_k$ be composite with a_k prime, $a_k \neq 2$ and $k \neq l \Rightarrow a_k \neq a_l$ Which *b* solve now $b^2 \equiv n \,(mod\, a)$?

From $b^2 \equiv n \,(mod \prod a_k)$ it immediately follows for all *k*: $b^2 \equiv n \,(mod\, a_k)$ and so for each a_k the same condition is valid as for the primes of the factor base $\left(\frac{n}{a_k}\right) = +1$ and thanks to Shanks-Tonelli we can compute the two solutions $\pm\beta_k$ of $\beta_k^2 \equiv n \,(mod\, a_k)$ for all *k*. With the help of the chinese remainder theorem, we are now able to construct our desired

$$b \equiv \sum \beta_k \cdot \bar{a}_k \cdot \frac{a}{a_k} \;\; (mod\, a) \quad \text{with} \quad \bar{a}_k \cdot \frac{a}{a_k} \equiv 1 \;\; (mod\, a_k)$$

where $\bar{a}_k$ is the multiplicative inverse of $\frac{a}{a_k}$ modulo a_k , which exists as a_k and $\frac{a}{a_k}$ are relatively prime (see invmod, a recursive definition of the extended Euklidian algorithm).

Assume *a* had $d+1$ prime factors, so $k \in \{0, 1, \ldots, d\}$. Due to the multiplicity of the β_k we get a total of 2^{d+1} different roots

$$b(\mu) \equiv \sum_{k=0}^{d} \mu_k \cdot b_k \;\; (mod\, a) \quad \text{with} \quad b_k := \beta_k \cdot \bar{a}_k \cdot \frac{a}{a_k} \quad \text{and} \quad \mu \in \{-1, +1\}^{d+1}$$

HMPQS

Consequently the HMPQS makes use of this fact, and a corresponding number of different polynomials $g_b(x)$ can be constructed for a composite *a* at once.

Consider now μ and ν with $\nu_k = -\mu_k$ for all $k \in \{0, 1, \ldots, d\}$. Obviously it is $b(\mu) = -b(\nu)$. Unfortunately we now have

$$g_{-b}(x) = (a \cdot x - b)^2 - n = (a \cdot (-x) + b)^2 - n = g_b(-x)$$

and as we sieve in an interval $x \in [-M\,; +M]$, we would effectively sieve the same numbers twice. To avoid this we need to prevent to alternate all signs μ_k and therefore we set $\mu_0 := +1$ and there are still 2^d different polynomials $g_b(x)$ remaining.

To obtain a large number of polynomials, *d* needs to be large as well, so *a* should have many small prime factors a_k and due to $\left(\frac{n}{a_k}\right)=+1$ this *a* is *B*-smooth and each a_k divides $g_b(x)$ for all $x\in\mathbb{Z}$.

There are two open questions. How can one compute the s_j for some $g_b(x)$? And finally, how do we choose *d* and the $a_k\in B$, so that $a=\prod_{k=0}^{d} a_k\approx a_{ideal}$?

The hypercube

In the paragraph "sieving" we defined the r_j with $r_j^2\equiv n\ (mod\ p_j)$. For the QS we needed to calculate them for all $p_j\in B$, due to their useful property $p_j \mid f(\pm r_j+z\cdot p_j)$ for all $z\in\mathbb{Z}$. The HMPQS requires them as β_k , because of $\beta_k=r_j$ for $a_k=p_j\in B$. Besides we will need them to calculate the s_j with the useful property $p_j \mid g(s_j+z\cdot p_j)$: Due to $g(x)=f(ax+b)$ it immediately follows that $a\cdot s_j+b\equiv r_j\ (mod\ p_j)$. To solve this congruence for s_j , we need to have the multiplicative inverse of *a* modulo p_j , let's say $\mathring{a}_j$ with $a\cdot\mathring{a}_j\equiv 1\ (mod\ p_j)$. This inverse exists only if *a* and p_j are relatively prime. No problem, because otherwise, we have $p_j=a_k$ and we know already that $p_j \mid g(x)$, that's after all the way we constructed $g(x)$ (ulimyhmpqs sets `factor-1/amodp` $\leftarrow -1$, to mark those $p_j=a_k$ that will not be sieved in). Finally we obtain two solutions

$$s_j\equiv \mathring{a}_j\cdot(\pm r_j-b)\ (mod\ p_j) \quad .$$

Initially we have to compute the $b(\mu)$ for each μ and the s_j for the complete factor base before we can start sieving. This computational effort is not to be underestimated! But there is another trick: Each μ with $\mu_0=+1$ corresponds to the corner of a *d*-dimensional hypercube of edge length 2 and the awkward sum-calculation for $b(\mu)$ becomes one simple addition if we just traverse from one corner to a neighbouring corner, in other words, change the sign of only one of the μ_k at a time. We reach each corner of the hypercube if *k* follows a Hamiltonian path:

path(d): if d=0 then [ ] else path(d-1) & [d] & path(d-1)

e.g. path(4)=[1 2 1 3 1 2 1 4 1 2 1 3 1 2 1]

Let $\mu'_k=-\mu_k$ and $\mu'_{i\neq k}=\mu_i$, then we obtain $b(\mu')\equiv b(\mu)-2\cdot\mu_k\cdot b_k\ (mod\ a)$. Especially we have $s_j(\mu')\equiv s_j(\mu)+\mu_k\cdot 2\mathring{a}_j b_k\ (mod\ p_j)$ and it is worth to store the values $2\mathring{a}_j b_k$ in a two-dimensional array (`kip`) in advance, before starting to traverse the hypercube.

The *a*-generator

How to construct a with $a=\prod_{k=0}^{d} a_k \approx a_{ideal}$ and $a_k \in B$? d shall be as large as practicable, so the chosen a_k need to be small. Remember, that $p_j = a_k$ cannot be sieved in. If we now select the smallest $p_j \in B$ as a_k , the resulting polynomials $h(x)$ generate probably less *B*-smooth numbers, as if we had chosen larger a_k . Additionally *d* should be dimensioned so, that it takes at least a few *a* to factor *n*, so it was worth to prepare the corresponding hypercubes (analogous to the argument for using sub-intervals for the QS).

Similar as described in [2] ulimyhmpqs therefore uses a_k with $a_k = p_j$ and $2q_{min} \leqslant j < 2q_{max}$. Within these boundaries there are $\bar{q} = q_{max} - q_{min}$ even and odd *j*. Now all $\binom{\bar{q}}{3}$ possible permutations of three different odd indices are generated:

$$top := p_{j_1} \cdot p_{j_2} \cdot p_{j_3} \quad \text{with} \quad j_1 \neq j_2 \neq j_3 \neq j_1 \quad .$$

So the smallest product will be $top_{min} = p_{2q_{min}+1} \cdot p_{2q_{min}+3} \cdot p_{2q_{min}+5}$. These triples are distributed among an array *A* of size *T* according to their logarithms, each place of the array may contain more than one element:

$$A[index(\log top)] \leftarrow top \quad \text{with} \quad index(x) := [\frac{x - \log top_{min}}{\log top_{max} - \log top_{min}} \cdot T] \quad .$$

Finally a recursive function systematically multiplies $a \leftarrow 1$ by factors with even indices until *a* is large enough, so that the triples in *A* can be taken to "top it off", to meet $a \cdot top \approx a_{ideal}$. Recursion starts with make-cubes $(1, q_{min})$.

```
make-cubes (a, q) :
        if  log a_ideal − log a > log top_min  then
                unless  log a_ideal − log a > log top_max
                        use-cube (a · A[index(log a_ideal − log_a)])
        else if  q < q_max  then
                make-cubes (a · p_2q, q+1)
                make-cubes (a, q+1)
```

This pseudo-code does not take into account, that a place of the array is either empty or contains more than one triple, so either no or a few hypercubes at once are generated. The author's intention was to demonstrate, that make-cube does not generate hypercubes of a fixed dimension *d* (it starts with the higher possible *d*) and that all generated *a* are different, because the intersection of the two subsets of *B* (even and odd *j*) is empty.

It required some experimenting to find reasonable parameters, so that on one hand the array is not too sparse (to assure a sufficient number of hypercubes is generated) but on the other hand large enough, so that the error

$$error(a) := |a - a_{ideal}| / a_{ideal}$$

becomes small. By setting $q_{min} = 5$ the first nine primes of the factor base will not be used at all. For $n \geqslant 10^{42}$ we set $q_{max} = 105 \ll \frac{\#B}{2}$ and achieve a total of 161 700 possible topoffcubes which we distribute among $T = 20\ 000$ places. This results in $error(a) < 0{,}05\%$, good enough to not have any significant drawback on the performance.

To prevent the number of generated hypercubes to be insufficient this way, for $n < 10^{42}$, the *top* will be made of only two factors to limit their size. Below $n \approx 10^{25}$ now q_{max} is limited also by $\#B/2$. This all leads to have less *tops*, which is counteracted by setting $T := min\{20\ 000, \lfloor \#\{top\}/8 \rfloor\}$. Nevertheless an $error(a) \approx 0{,}5\%$ will be achieved and tests with 10^5 different $n \approx 10^{23}$ always generated sufficient *a*. For $n < 10^{23}$ ulimyhmpqs diverts to the rigorous algorithm make-cublets, that uses all elements of *B* regardless of *error(a)*. This was just to close the gap to probe-division, which is sufficient to factor numbers up to $n \approx 10^{12}$ if one has the first 100 000 primes at hand (Of course there are better algorithms for numbers of intermediate size).

The composition of the factor base can be assumed to be random, the probability for $\left(\frac{n}{p}\right) = +1$ is roughly $\frac{1}{2}$. So there is a very small risk that make-cubes fails. If so, the *a*-generator reverts to use two- instead three-factor *tops* and finally continues by using make-cublets to finish the factorization.

The sieve in detail

To refer to the sieve array more efficiently, the intervall $X := [-M\,;M]$ is projected onto $Y := [0\,;2\mathrm{M}]$. The resulting sieve polynomial is

$$h'(y) = h(y - M) = (a\,y + \beta) \cdot y + \gamma$$

$$\text{with } \beta := 2(b - Ma) \text{ and } \gamma := aM^2 - 2\mathrm{bM} + \frac{b^2 - n}{a} \ ,$$

So two bignum multiplications and additions are necessary to compute an individual value $h'(y)$. Initializing the sieve by $S_y \leftarrow \lfloor \log_2 |h'(y)| \rfloor$ would be much too slow, if we had to compute each $h'(y)$ within $y \in [0\,;2\mathrm{M}]$. Alternatively ulimyhmpqs determines sub-intervals $Y_l \subset Y$ for $l \in \mathbb{N}$, so that $\lfloor \log_2 |h'(y)| \rfloor = l$ for all $y \in Y_l$ Unfortunately the parabola has four such sub-intervals for each *l*, according to the signs of $h'(y)$ and $y - \frac{\beta}{2\mathrm{a}}$. Besides the inverse function of $h'(y)$ is required to determine the boundaries $h'^{-1}(2^l)$ of the sub-intervals. For performance reasons floating point arithmetic is used to take the required roots.

Finally the factors p_j with $p_j \nmid a$ will be sieved in. Of course both "zeros" s_j are to be considered. With $p_j | g(s_j)$ we get $p_j | h'(s_j + M)$ and starting with the smallest $y \in Y$ with $p_j | h'(y)$ a loop runs across the sieve:

$$y \leftarrow s_j + M + \left[\frac{-M - s_j}{p_j}\right] \cdot p_j$$

repeat $\quad S_y \leftarrow S_y - \lfloor \log_2 p_j \rfloor$

$\quad y \leftarrow y + p_j$

until $\quad y > 2M$

Thereafter, when scanning the sieve for values $S_y <$ threshold, all p_j have to be considered for probe-dividing, even those with $p_j | a$, as $g(y - M) = a \cdot h'(y)$ might be divisible by a power of a_k, especially as the a_k were chosen to be small. If probe-division now reveals a $h'(y_i)$ to be *B*-smooth,

$$h'(y_i) = h(y_i - M) = \prod p_j^{e_{i,j}}$$

a new relation has been found, because

$$g(y_i - M) = a \cdot h(y_i - M) = (a(y_i - M) + b)^2 - n$$

will be *B*-smooth as well. Instead of multiplying by *a* and dividing again by a_k thereafter, the new relation can be expressed as

$$\langle \{i\} ; \{j \mid e_{i,j} \bmod 2 = 1\} * \{j \mid (\exists k)\ (a_k = p_j)\} \rangle \quad \text{and} \quad x_i := a(y_i - M) + b$$

and processed as described above (solving).

Summary of the algorithm

The sieving part of the algorithm has been summarized on the next page in pseudo code with some references to the source-code. The solving part and the *a*-generator have been described above, and will not be repeated here.

Given *n*, #*B* and *M*, the factor base $B := \{-1, p_1, p_2, \ldots\}$ will be set up like this:

set $j \leftarrow 2$, $p_0 \leftarrow -1$, $p_1 \leftarrow 2$

for $p \leftarrow 3,5,7,11...$ (the odd primes)

if $\left(\frac{n}{p}\right) = +1$ then set $p_j \leftarrow p$, $j \leftarrow j + 1$

until $j = \# B$

For all $p_j \in B \setminus \{-1\}$ we compute $r_j \leftarrow \sqrt{n} \,(mod\ p_j)$ and $\lfloor \log_2 p_j \rfloor$ in advance. Let $i \leftarrow 0$ the number of found relations. To prepare the single large prime extension, we further need an array *H* of 2^h places, each initialized to $H_\tau \leftarrow nil$, which serves to store partial relations. Full relations will be stored in x_i (xarray) and the buckets described in the paragraph on solving.

For each generated hypercube $a = a_0 \cdot a_1 \cdots a_d \approx a_{ideal}$ with $a_k = p_{j_k}$ (a-generator)

for $k \leftarrow 0,\ldots,d$ set $\bar{a}_k \leftarrow (\frac{a}{a_k})^{-1} (mod\ a_k)$, $b_k \leftarrow r_{j_k} \cdot \bar{a}_k \cdot \frac{a}{a_k}$

set $b \leftarrow nil$

for $k \leftarrow 1,2,1,2,3,2,1,....$ (Hamiltonian path, b-generator)

if $b = nil$ then set $b \leftarrow \sum_{k=0}^{d} b_k$

for $p_j \in B' := \{p | (p \geq \text{omit-below}) \wedge (\neg p | a)\}$

set $\mathring{a}_j \leftarrow \frac{1}{a} (mod\ p_j)$

set $s_j \leftarrow \mathring{a}_j \cdot (+r_j - b)\ (mod\ p_j)$

set $s_j' \leftarrow \mathring{a}_j \cdot (-r_j - b)\ (mod\ p_j)$

for $k' \leftarrow 1,\ldots,d$ set $\text{kip}_{j,k'} \leftarrow 2\mathring{a}_j b_{k'}$, $\mu_k \leftarrow 1$

else set $b \leftarrow b - 2 \cdot \mu_k \cdot b_k$

for $p_j \in B'$

set $s_j \leftarrow s_j + \mu_k \cdot \text{kip}_{j,k}\ (mod\ p_j)$

set $s_j' \leftarrow s_j' + \mu_k \cdot \text{kip}_{j,k}\ (mod\ p_j)$

set $\mu_k \leftarrow -\mu_k$

set $\beta \leftarrow 2(b - Ma)$, $\gamma \leftarrow aM^2 - 2\text{bM} + \frac{b^2 - n}{a}$ (sieve-for-ab)

let $h'(y) := (a\,y + \beta) \cdot y + \gamma$

for $l \leftarrow 0,1,2...$ (init-sieve)

for each $Y_l \subset [0;2\text{M}]$ with $y \in Y_l \Rightarrow \lfloor \log_2 h'(y) \rfloor = l$

for $y \in Y_l$ set $S_y \leftarrow l$

for $p_j \in B'$ (sieve-in)

set $y \leftarrow s_j + M + \lceil \frac{-M - s_j}{p_j} \rceil \cdot p_j$

while $y \leq 2\text{M}$ set $S_y \leftarrow S_y - \lfloor \log_2 p_j \rfloor$, $y \leftarrow y + p_j$

set $y \leftarrow s_j' + M + \lceil \frac{-M - s_j'}{p_j} \rceil \cdot p_j$

while $y \leq 2\text{M}$ set $S_y \leftarrow S_y - \lfloor \log_2 p_j \rfloor$, $y \leftarrow y + p_j$

for $y \leftarrow 0,\ldots,2M$

if $S_y < \text{threshold}$ then (divide-out-mod-2)

set $R \leftarrow h'(y)$, $x \leftarrow a(y - M) + b$, $J \leftarrow \{j_{0,\ldots}, j_d\}$

if $R < 0$ then set $R \leftarrow -R$, $J \leftarrow J * \{0\}$

for $p_j \in B \setminus \{-1\}$

set $e \leftarrow 0$

while $R\ mod\ p_j = 0$ set $e \leftarrow 1 - e$, $R \leftarrow R / p_j$

if $e = 1$ then set $J \leftarrow J * \{j\}$

if $R = 1$ then set $i \leftarrow i + 1$, $x_i \leftarrow x$

put-into-its-bucket $(\{i\}, J)$

else

set $\tau \leftarrow \lfloor R/2 \rfloor mod\ 2^h$

if $H_\tau = nil$ then set $H_\tau \leftarrow (R, x, J)$

else

let $(R', x', J') := H_\tau$

if $R = R'$ then

set $i \leftarrow i + 1$, $x_i \leftarrow x \cdot x'$

put-into-its-bucket $(\{i\}, J * J')$

else if $R < R'$ then set $H_\tau \leftarrow (R, x, J)$

Parameter optimizing

A variety of parameters need to be adjusted, to optimize run time. These are the size of the factor base *#B*, the sieve size *2M*, the limit omit-below (primes smaller than this will not be sieved in), the size of the hash-table to keep partial relations, and the sieve threshold. ulimyhmpqs allows to specify these parameters manually, but though it offers some reasonable default values. When determining these default values empirically, seven problems occured.

First, there are dependencies between parameters (e.g. a larger omit-below should go with a larger threshold) which requires multi-dimensional test series. Second, due to the almost-exponential run time characteristics it is difficult to conclude from test results for smaller *n* to higher *n*, so especially when we need good parameters, because run-time is extremely high, we can't run complete tests. Third, optimum parameters for some *n* may differ from another *n* of almost the same size due to the quasi-random composition of the factor base. Fourth, if you try to substitute test series by prediction through math, you enter a difficult, apparently only partly explored area of analytic number theory. Sixth, the nature of LISP garbage collection causes noise in time measurement. Seventh, parameters depend on the individual machine, the used LISP and must be limited so that the required memory does not exceed the available.

The most important parameter is the factor base size *#B*. Fortunately there is this heuristic $\#B_{opt} \approx e^{\sqrt{2}/4 \cdot \sqrt{\ln n \cdot \ln \ln n}}$ which ulimyhmpqs uses, although in almost every case 75% of this value runs faster. This was decided, as especially when *n* turns out to have a very unfavourable factor base (that lacks small factors), *#B* has to be higher to compensate. I have started to investigate heuristic estimates for the frequency of smooth numbers of given size that do not ignore the composition of the factor base (like in [1] and [4]) but this work is not finished yet. Finally the hmpqs needs $\sim \#B^2$ memory, which becomes a bottleneck, when factoring large numbers. The default-function factor-base-size limits #B to 9 000, which will require roughly 100MB.

Thereafter test series with different *M* and *n* up to 10^{60} were run, and the statistic optimum was approximated by $M_{opt} \approx 4800\, n^{.0182}$ (see m-opt).

Another test series showed, that it is in average favourable to not sieve in primes below 25 (omit-below), if the threshold is increased by $\frac{1}{2} \cdot \log_2 p$ for each prime *p* that is omitted.

When trying to factor numbers greater than 10^{60} with ulimyhmpqs two aspects have to be considered: First *#B* has to be chosen lower than $\#B_{opt}$, because the gaussian reduction will become slow. Besides the available memory has to be taken care of. Second the threshold has to be increased (delta-t) in order to catch more partial relations, they will now play an important role.

The following table of average run-times refers to ulimyhmpqs running on Macintosh Common LISP 4.3.5. on a 500MHz G3 processor. Except for the manual setting of *#B*=7000 for $n \approx 10^{60}$ the default parameters were used.

$\log_{10} n$	*#B*	*M*	time
30	420	16900	1.3s
40	1350	25700	23s
50	3800	39000	270s
60	7000	59000	3800s

4998267725099435687608264798548361735594197211550916814863113089903531 was the 70-digit number, and $n = 2^{256} + 1$ the 78-digit number used in the following test runs. Where in the table above in average less than 10% of the relations where found *due* to partials, this rate could be increased by choosing a higher threshold (the number in parentheses refers to the `delta-t` parameter).

$\log_{10} n$	*#B*	memory	*M*	threshold	*due*	time
70	9000	100MB	80000	22 (+0)	14%	38986s
				27 (+5)	34%	32402s
				30 (+8)	45%	28590s
77	20000	500MB	122257	31 (+10)	47%	263300s

For the last two tests the size of the hash-table had been precautionary changed from the default setting of 2^{16} to 2^{20} to avoid collisions, which turned out to be sufficient, as only 12% of the hash-table places were occupied after the last 73hrs test.

The source code contains a description of the parameters and the functional access to ulimyhmpqs, as well as a description of the progress report, that will be displayed if the `:report` parameter is set to `T`. Users who like to optimize ulimyhmpqs for large numbers will find some code, that allows to gather performance data without actually completing a factorization.

References

One will find a lot to read about the quadratic sieve. At this place only papers that have been referred to are listed. I personally recommend to read what its inventor Carl Pomerance himself wrote about the QS (e.g. [1] and [3], the latter also giving a brief description to the number field sieve). The probably best source of help for implementors currently found in the internet is [4], which unfortunately was published after ulimyhmpqs had been developed.

[1] C. Pomerance

Smooth numbers and the quadratic sieve

to appear in the proceedings of an MSRI workshop, J.Buhler and P. Stevenhagen, eds.

http://www.dartmouth.edu/~carlp/PDF/qstalk3.pdf

[2] B. Carrier, S. Wagsta

Implementing the Hypercube Quadratic Sieve with Two Large Primes

Proceedings of the International Conference on Number Theory for Secure Communications, 2003.
https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/2001-45.pdf

[3] C. Pomerance

A tale of two sieves

The notices of the AMS 43 (1996) 1473-1485

http://www.ams.org/notices/199612/pomerance.pdf

[4] M. Kechlibar

The Quadratic Sieve - introduction to theory with regard to implementation issues

http://www.karlin.mff.cuni.cz/~krypto/mpqs/main_file.pdf