FactInt
—
A GAP4 Package

Version 1.2

by
Stefan Kohl

E-mail: kohl@mathematik.uni-stuttgart.de

This package for GAP 4 provides routines for integer factorization, in particular:

Pollard's $p - 1$

Williams' $p + 1$

The Elliptic Curves Method (ECM)

The Continued Fraction Algorithm (CFRAC)

The Multiple Polynomial Quadratic Sieve (MPQS)

It is completely written in the GAP language and contains / requires no external binaries. No other packages are needed. It must be installed in the `pkg` subdirectory of the GAP distribution.

Contents

1 Preface

Factoring large integers is a computationally very difficult problem, and there is no general factorization algorithm that is capable of factoring arbitrary integers with more than about $130 - 150$ decimal digits on currently existing computers, but however, there are methods (not algorithms in the sense that it is guaranteed that they will give the desired result after a finite number of steps) for factoring integers with prime factors being far beyond the range where trial division is feasible.

One important class of such methods is based on exponentiation in suitably chosen groups acting on subsets of the k-fold cartesian product of the set of residue classes (mod n), where n denotes the number to be factored. Representatives of this class are the Elliptic Curves Method (ECM), Pollard's $p - 1$ and Williams' $p + 1$. These methods have the important property that they find smaller factors usually considerably faster than larger ones (but with the drawback of being slower for large factors).

The other important class consists of the so-called factor base methods. These methods compute factorizations of perfect squares (mod n) over an appropriately chosen factor base (a set of small prime numbers, or of prime ideals in the case of the Generalized Number Field Sieve) and use them to determine a pair of integers (x, y), such that x^2 and y^2 are congruent (mod n), but $\pm x$ and $\pm y$ are not. In this situation, taking $\mathrm{Gcd}(n, x - y)$ will yield a non-trivial factor of n. Representatives of this class are the Continued Fraction Algorithm (CFRAC), the Multiple Polynomial Quadratic Sieve (MPQS) and the already mentioned Generalized Number Field Sieve (GNFS), which has got the asymptotically lowest complexity of all factoring methods known today (in respect of "difficult" numbers, of course), with the drawback of being superior to the MPQS only for numbers with more than 100 decimal digits, say, which is probably not within the feasible range of such a function implemented in GAP. The first two methods are implemented in this package.

The only method which I am aware of (except of the "naive" ones like trial division and some "historical" methods) that does not fit into one of the two above-mentioned classes is Pollard's Rho, which is already implemented in the GAP Library.

In respect of the current state-of-the-art in integer factorization, see for example the respective web pages of the RSA Laboratories, e.g.

> http://www.rsasecurity.com/rsalabs/challenges/factoring/numbers.html

I would like to thank Bettina Eick and Steve Linton for their support and many interesting discussions.

I would be grateful for any bug reports, comments or suggestions,

<div align="right">Stefan Kohl</div>

2 The General Factorization Routines

2.1 If you do not care about the methods used

1 ▸ IntegerFactorization(n)   F

IntegerFactorization computes the prime factorization of the integer n. The result is returned as a list of the prime factors. In case of failure an error is signalled.

The returned factors are considered to be prime by the built-in probabilistic primality test of GAP (IsProbablyPrimeInt, see the Reference Manual), as in all other factorization routines included in this package.

IntegerFactorization(n) is equivalent to FactInt(n : FactIntPartial:=false)[1] (see 2.2.1).

IntegerFactorization is also installed as a method for Factors, with a higher value than the GAP library function FactorsInt.

```
gap> IntegerFactorization( Factorial(24) - 1 );
[ 625793187653, 991459181683 ]

gap> Factors( 1459^24 - 1 );
[ 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 5, 7, 13, 73, 97, 193, 283, 337, 1009,
  303889, 669433, 1064341, 6722971513, 4531280671081, 313380751265929 ]
```

2.2 Taking influence on the methods being used

1 ▸ FactInt(n)   F

FactInt computes the prime factorization of the integer n. The result is returned as a list of two lists, where the first one contains the prime factors found, and the second one contains remaining unfactored parts of n, if there are any (if *FactIntPartial* not set, this should never happen, because in this case the function does not return before n is factored completely).

The factors returned as prime factors are considered to be prime by the build-in probabilistic primality test of GAP (IsProbablyPrimeInt, see the Reference Manual). This also applies to all other factorization routines provided by this package.

FactInt recognizes the following options:

TDHints
> specifies a list of additional trial divisors (when factoring "random" numbers, giving a large list of trial divisors here is certainly not a sensible approach, *TDHints* is useful only if certain primes p are expected to divide n with probability significantly larger than $\frac{1}{p}$)

RhoSteps
> specifies the number of steps for Pollard's Rho,

RhoCluster
> specifies the number of steps between two Gcd computations in Pollard's Rho

Pminus1Limit1

> specifies the first stage limit for Pollard's $p - 1$ (see 3.1.1)

Pminus1Limit2

> specifies the second stage limit for Pollard's $p - 1$ (see 3.1.1)

Pplus1Residues

> specifies the number of residues to be tried by Williams' $p + 1$ (see 3.2.1)

Pplus1Limit1

> specifies the first stage limit for Williams' $p + 1$ (see 3.2.1)

Pplus1Limit2

> specifies the second stage limit for Williams' $p + 1$ (see 3.2.1)

ECMCurves

> specifies the number of elliptic curves to be tried by the Elliptic Curves Method (ECM) (see 3.3.1), also admissible: a function that takes an argument n (the number to be factored) and returns the desired number of curves to be tried

ECMLimit1

> specifies the initial first stage limit for ECM (see 3.3.1)

ECMLimit2

> specifies the initial second stage limit for ECM (see 3.3.1)

ECMDelta

> specifies the increment per curve for the first stage limit in ECM, the second stage limit is adjusted appropriately (see 3.3.1)

ECMDeterministic

> if true, ECM chooses its curves deterministically, i.e. repeatable (see 3.3.1)

FactIntPartial

> if true, the partial factorization obtained by applying the factoring methods whose time complexity depends mainly on the size of the factors to be found and less on the size of n (those of the first class mentioned in the preface) is returned and the factor base methods (MPQS and CFRAC) are not used to complete the factorization for numbers that exceed the bound given by *CFRACLimit* resp. *MPQSLimit*; default: false

FBMethod

> specifies which of the factor base methods should be used to do the "hard work"; currently implemented: CFRAC and MPQS (see 3.4.1, 3.5.1)

CFRACLimit

> specifies the maximal number of decimal digits of an integer to which the Continued Fraction Algorithm (CFRAC) should be applied (only used when *FactIntPartial* = true) (see 3.4.1)

MPQSLimit

> as above, for the Multiple Polynomial Quadratic Sieve (MPQS) (see 3.5.1)

Once these option values are set by the user via `PushOptions`, all subsequent factorizations by `FactInt` and `IntegerFactorization` (see 2.1.1) are done using these settings without the need to give further arguments (in respect to the use of the GAP Options Stack, see the Reference Manual). Setting *RhoSteps*, *Pminus1Limit1*, *Pplus1Residues*, *Pplus1Limit1*, *ECMCurves* or *ECMLimit1* to zero switches the respective method off. `FactInt` chooses defaults for all option values that are not explicitly set by the user. The option values are also taken into consideration by the routines for the particular factorization methods described in the next chapter when default values have to be chosen.

If $|n| < 10^{12}$, `FactInt` just calls the library function `FactorsInt`, which is guaranteed to give the correct result for numbers in this range. If not, it checks whether $n = b^k \pm 1$ for some b, k and looks for factors

corresponding to polynomial factors of $x^k \pm 1$ (in order to immediately obtain factors that do not correspond to polynomial factors it is necessary to give a list of the prime factors of numbers of this form as *TDHints*).

As the first general factorization step, `FactInt` does trial division by the primes below 1000. After that, it checks whether the remaining part of the number to be factored already is a prime. If not, it does trial divisions by some already known primes, using the list `Primes2` (see GAP-Manual). If there is still an unfactored part m, then it checks whether m is a non-trivial power of an integer. Then, the additional list of trial divisors given as *TDHints* is processed, if present.

After the small and other "easy" factors have been casted out this way, `FactInt` searches for "medium-sized" factors using Pollard's Rho (by the library function `FactorsRho`, see GAP-Manual), Pollard's $p - 1$ (see 3.1.1), Williams' $p + 1$ (see 3.2.1) and the Elliptic Curves Method (ECM, see 3.3.1) in this order. After that, if there is still an unfactored part remaining and *FactIntPartial* = false (or the remaining composites do not exceed the bound given by *CFRACLimit* resp. *MPQSLimit*), one of the factor base methods (CFRAC, see 3.4.1 or MPQS, see 3.5.1) is used to do the "hard work", depending on the value of *FBMethod*, which could be `"MPQS"` or `"CFRAC"`.

Finally, it is checked whether the product of all factors is equal to n and whether all factors in the first list of the result pass the GAP pseudoprime test `IsProbablyPrimeInt` and all factors in the second list are really composites. These checks are also done by all other factorization routines provided by this package.

```
gap> FactInt( Factorial(39) + 1 : RhoSteps := 16384, Pminus1Limit1 := 100000,
             Pplus1Limit1 := 2000, ECMLimit1 := 5000, ECMCurves := 10,
             FBMethod := "MPQS" );
[ [ 79, 57554485363, 146102648914939, 30705821478100704367 ], [  ] ]
```

3 The Routines for Specific Factorization Methods

Descriptions of the factoring methods used here can be found in [Bre89] and in [Coh93]. The last book contains also descriptions of the other methods mentioned in the preface.

3.1 Pollard's p-1

1 ▶ FactorsPminus1(n , [[a], *Limit1*, [*Limit2*]])   F

FactorsPminus1 tries to factor n using Pollard's $p - 1$, with a as base for exponentiation (the default is $a = 2$), *Limit1* as first stage limit and *Limit2* as second stage limit. FactorsPminus1 chooses defaults for all arguments not explicitly given. If FactorsPminus1 is called with three arguments, these arguments are regarded as n, *Limit1* and *Limit2*. The result is returned as a list of two lists, where the first one contains the prime factors found, and the second one contains remaining unfactored parts of n, if there are any.

Pollard's $p - 1$ is based on the fact that exponentiation in the group of invertible residue classes (mod n) is so fast that it is possible to compute for example $a^{k!}$ for k large enough (e.g. 100000 or so) in a reasonable amount of time and without using much memory, and on Lagrange's Theorem, which states that $a^{k!}$ is congruent to 1 (mod p) (where p is a prime divisor of n) if $k!$ is a multiple of $p - 1$, and $(a, p) = 1$. In this situation, if this is satisfied for no other prime factor of n, p can be determined by calculating $Gcd(a^{k!} - 1, n)$. A prime divisor p is usually found if the largest prime factor of $p - 1$ is not larger than *Limit2*, and the second-largest one is not larger than *Limit1*. (Compare with 3.2.1 and 3.3.1.)

```
gap> FactorsPminus1( Factorial(158) + 1, 100000, 1000000 );
[ [ 2879, 5227, 1452486383317, 9561906969931, 18331561438319,
      48371429970948376081158111034173295050649321812265485340067492134508231 0\
9063704522956548165713050412173230528798429248261213331432547136748329627731 07\
8067899457155703860385652567196145249247051651100481487161609649806290811760 57\
0095669 ], [  ] ]

# Let's see why this works:

gap> List( last[ 1 ]{[ 3, 4, 5 ]}, p -> IntegerFactorization( p - 1 ) );
[ [ 2, 2, 3, 3, 81937, 492413 ], [ 2, 3, 3, 3, 5, 7, 7, 1481, 488011 ],
  [ 2, 3001, 7643, 399613 ] ]
```

3.2 Williams' p+1

1 ▶ `FactorsPplus1( ` n `, [ [ ` *Residues* ` ], ` *Limit1* `, [ ` *Limit2* ` ] ] )`   F

`FactorsPplus1` tries to factor n using a variant of Williams' $p+1$, where it tries *Residues* different residues (the probability of a given residue to be in principle usable is about $1/2$) and uses *Limit1* as first stage limit and *Limit2* as second stage limit. `FactorsPplus1` chooses defaults for all arguments not explicitly given. If `FactorsPplus1` is called with three arguments, these arguments are regarded as n, *Limit1* and *Limit2*. The result is returned as a list of two lists, where the first one contains the prime factors found, and the second one contains remaining unfactored parts of n, if there are any.

Williams' $p+1$ is very similar to Pollard's $p-1$ (see 3.1.1), the only difference is that the group used here has order $p+1$ (instead of $p-1$), and that the group operation takes more time. A prime divisor p is usually found if the largest prime factor of $p+1$ is at most *Limit2* and the second-largest one is not larger than *Limit1*, and if the algorithm hits a "usable" residue (concerning the "unusable" residues, it could be stated that the order of the respective group is $p-1$, such that the method turns into a slow $p-1$-algorithm in this case). (Compare also with 3.3.1.)

```
gap> FactorsPplus1( Factorial(55) - 1, 10, 10000, 100000 );
[ [ 73, 39619, 277914269, 148257413069 ],
  [ 1065435291200499549550850766345372624597189863957 ] ]
gap> List( last[ 1 ], p -> [ Factors( p - 1 ), Factors( p + 1 ) ] );
[ [ [ 2, 2, 2, 3, 3 ], [ 2, 37 ] ],
  [ [ 2, 3, 3, 31, 71 ], [ 2, 2, 5, 7, 283 ] ],
  [ [ 2, 2, 2207, 31481 ], [ 2, 3, 5, 9263809 ] ],
  [ [ 2, 2, 47, 788603261 ], [ 2, 3, 5, 13, 37, 67, 89, 1723 ] ] ]
```

3.3 The Elliptic Curves Method (ECM)

1 ▶ `FactorsECM( ` n `, [ ` *Curves* `, [ ` *Limit1* `, [ ` *Limit2* ` [ ` *Delta* ` ] ] ] ] )`   F

`FactorsECM` tries to factor n using the Elliptic Curves Method (ECM), where *Curves* specifies the number of curves to be tried, *Limit1* is the initial first stage limit, *Limit2* is the initial second stage limit and *Delta* is the increment per curve for the first stage limit, where the second stage limit is adjusted appropriately. `FactorsECM` chooses defaults for all arguments not explicitly given. The option *ECMDeterministic* specifies, if set, that the choice of the curves to be tried should be deterministic, i.e. that repeated calls of `FactorsECM` yield the same curves, and hence for the same n the result after the same number of trials (this is of use mainly for testing purposes). The result is returned as a list of two lists, where the first one contains the prime factors found, and the second one contains remaining unfactored parts of n, if there are any.

The Elliptic Curves Method is based on the fact that exponentiation in the elliptic curve groups $E(a,b)/n$ is fast enough that it is possible to compute for example $g^{k!}$ for k large enough (e.g. 10000 or so) in a reasonable amount of time and without using much memory, and on Lagrange's Theorem, which states that for each elliptic curve point g, $g^{k!}$ is the identity element of $E(a,b)/p$ (where p is a prime divisor of n) if $k!$ is a multiple of $|E(a,b)/p|$. In this situation, under reasonable circumstances, p can be determined by taking an appropriate Gcd.

In practice, the algorithm chooses in some sense "better" products P_k of small primes rather than $k!$ as exponents, and, after reaching the first stage limit with P_{Limit1}, it considers further products $P_{Limit1}q$ for primes q up to the second stage limit *Limit2* (which is usually set equal to, for example, 40 times the first stage limit, or so). The prime q corresponds to the largest prime factor of the order of the group under consideration.

A prime divisor p is usually found if the largest prime factor of the order of one of the examined elliptic curve groups $E(a,b)/p$ is at most *Limit2*, and the second-largest one is at most *Limit1*, so trying a larger number of curves increases the chance of factoring n as well as taking a larger value for *Limit1* and/or

Limit2 (it turns out to be not optimal either to take a large number of curves with very small *Limit1* and *Limit2* or to grind on with a single curve and very large limits). (Compare with 3.1.1.)

The elements of the group $E(a, b)/n$ are the points (x, y) given by the solutions of $y^2 = x^3 + ax + b$ in the residue class ring (mod n), and an additional point ∞ at infinity, which serves as identity element. To turn this set into a group, define the product (although elliptic curve groups are usually written additively, I prefer using the multiplicative notation here to retain the analogy to 3.1.1 and 3.2.1) of two points p_1 and p_2 as follows: Let l be the line through p_1 and p_2 if $p_1 \neq p_2$, otherwise let l be the tangent to the curve C given by the above equation in the point $p_1 = p_2$. l intersects C in a third point, say p_3 (if l does not intersect the curve in a third affine point, then set $p_3 := \infty$). Set $p_1 \cdot p_2$ equal to the image of p_3 under the reflection across the x-axis. Define the product of any curve point p and ∞ by p itself. This (more or less obviously, checking associativity requires some calculation) turns the set of points on the given curve into an abelian group $E(a, b)/n$.

However, the calculations are done in projective coordinates to have an explicit representation of the identity element and to avoid calculating inverses (mod n) for the group operation, which would otherwise require using an $O((logn)^3)$-algorithm, while multiplication (mod n) is only $O((logn)^2)$. The respective equation in this case is given by $bY^2Z = X^3 + aX^2Z + XZ^2$ (this form allows more efficient calculations than the Weierstrass model $Y^2Z = X^3 + aXZ^2 + bZ^3$, which is the projective equivalent to the affine representation $y^2 = x^3 + ax + b$ mentioned above). The algorithm only keeps track of two of the three coordinates, namely X and Z. The choice of curves is done in a way that ensures the order of the respective group to be divisible by 12. This increases the chance that it is smooth enough to find a factor of n. The implementation follows the description of R. P. Brent given in [Bre96], pp. 5 – 8 (in terms of this paper, for the second stage the "improved standard continuation" is used).

```
gap> FactorsECM(2^256+1,100,10000,1000000,100);
[ [ 1238926361552897,
      93461639715357977769163558199606896584051237541638188580280321 ], [  ] ]
```

3.4 The Continued Fraction Algorithm (CFRAC)

1 ▶ FactorsCFRAC(n)   F

FactorsCFRAC tries to factor n using the Continued Fraction Algorithm (CFRAC), also known as Brillhart-Morrison Algorithm. The result is returned as a list of the prime factors of n. In case of failure an error is signalled.

To CFRAC, the same warning applies as to the Quadratic Sieve (see 3.5.1).

The Continued Fraction Algorithm tries to find integers x, y, such that $x^2 \equiv y^2 \pmod{n}$, but not $\pm x \equiv \pm y \pmod{n}$. In this situation, taking $Gcd(x - y, n)$ yields a non-trivial factor of n. For determining such a pair (x, y), the algorithm uses the continued fraction expansion of the square root of n (because n is usually very large, it is impossible to compute the whole period of it, but a much smaller number of terms is enough in this case). If x_i/y_i is a continued fraction approximation for the square root of n, then $c_i := x_i^2 - ny_i^2$ has size only about a small constant times the square root of n. The algorithm tries to find as many c_i as possible which factor completely over a chosen factor base (a list of small primes) or with only one factor not in the factor base. The latter ones are usable only if a second c_i with the same "large factor" is found Then, Gaussian Elimination over $GF(2)$ is used to determine which of the congruences $x_i^2 \equiv c_i \pmod{n}$ have to be multiplied together to get a congruence of the desired form $x^2 \equiv y^2 \pmod{n}$, where the involved matrix M is given by $M_{ij} = 1$, if an odd power of the j-th element of the factor base divides the i-th usable factored value, and $M_{ij} = 0$ otherwise. For this purpose it is necessary that the number of factored c_i is larger than the rank of M, which is approximately given by the size of the factor base. (Compare with 3.5.1.)

```
gap> FactorsCFRAC( Factorial(34) - 1 );
[ 10398560889846739639, 28391697867333973241 ]
```

3.5 The Multiple Polynomial Quadratic Sieve (MPQS)

1 ▶ `FactorsMPQS( `n` )`   F

`FactorsMPQS` tries to factor n using the Single Large Prime Variation of the Multiple Polynomial Quadratic Sieve (MPQS). The result is returned as a list of the prime factors of n. In case of failure an error is signalled.

The intermediate results of a computation could be saved by interrupting the calculation with [Ctrl][C] and calling `Pause();` from the break loop. `FactorsMPQS` then pushes all data important for resuming the computation again as a record *MPQSTmp* on the options stack. When called again, it will look whether there is such an appropriate record on the options stack and get the previously computed factorization data, if so. For continuing the factorization process in another session, you will have to write this record to a file. This is done by the function `SaveMPQSTmp( `*filename*` )`, the file written by this function is readable by the standard `Read`-function of GAP.

Caution: The runtime of `FactorsMPQS` depends only on the size of n, not on the size of its factors, so if a small factor is not found during the preprocessing which is done before invoking the sieving process, you will have to wait as long as if n would have two prime factors of roughly equal size.

The Multiple Polynomial Quadratic Sieve tries to find integers x, y, such that $x^2 \equiv y^2 \pmod{n}$, but not $\pm x \equiv \pm y \pmod{n}$. In this situation, taking $Gcd(x - y, n)$ yields a non-trivial factor of n. For determining such a pair (x, y), the algorithm chooses polynomials f_a of the form $f_a(r) = ar^2 + 2br + c$ with suitably chosen coefficients a, b and c, satisfying $b^2 \equiv n \pmod{a}$ and $c = (b^2 - n)/a$. The identity $a \times f_a(r) = (ar + b)^2 - n$ yields a congruence (mod n) with a perfect square on one side and $a \times f_a(r)$ on the other. The algorithm searches for factorizations of polynomial values $f_a(r)$ over a chosen factor base (a list of small primes), either complete or with a single large factor which is not in the factor base (where the latter ones are only usable if a second $f_a(r)$ with the same "large factor" is found) using a sieving technique over an appropriately chosen sieving interval (similar to the Sieve of Eratosthenes). Taking more polynomials and hence shorter sieving intervals gives the advantage of having smaller $f_a(r)$ to factor over the factor base. Then, Gaussian Elimination over $GF(2)$ is used to determine the congruences which have to be multiplied together to get a congruence of the desired form $x^2 \equiv y^2 \pmod{n}$, where the involved matrix M is given by $M_{ij} = 1$, if an odd power of the j-th element of the factor base divides the i-th usable factored value, and $M_{ij} = 0$ otherwise. For this purpose it is necessary that the number of factored $f_a(r)$ is larger than the rank of M, which is approximately given by the size of the factor base. (Compare with 3.4.1.)

```
gap> FactorsMPQS( Factorial(38) + 1 );
[ 14029308060317546154181, 37280713718589679646221 ]
```

4 Getting Information about the Factoring Process

1 ▶ IntegerFactorizationInfo V

is an Info class (see 7.4 in the GAP Reference Manual for a description of the Info mechanism) that provides a way of seeing what's going on during the factoring process.

If `InfoLevel(IntegerFactorizationInfo) = 1`, then basic information about the factoring techniques used is displayed. If this InfoLevel has value 2, then additionally all "relevant" steps in the factoring algorithms are mentioned, and if it is set to 3, then large amounts of details of the progress of the factoring process are shown.

For convenience,

2 ▶ FactInfo(*level*) F

sets the `InfoLevel` of `IntegerFactorizationInfo` to the positive integer *level*. In other words,

```
FactInfo( level );
```

is equivalent to:

```
SetInfoLevel( IntegerFactorizationInfo, level );
```

The informational output is not guaranteed to be literally the same in each factorization attempt to a given integer with given parameters.

The timings in the example are given for a Pentium 200.

```
gap> FactInfo(2);
gap> Factors(1459^60-1);
#I
#I  Check for n = b^k +/- 1
#I
69713600543951832133936434146273147964393116471833882216998576159853318415596 8\
68768326730907183176778305329466153330965334450159052785791168346065270445439 2\
21141737129453841799132142733244004 00 = 1459^60 - 1
#I  The factors corresponding to polynomial factors are
[ 1458, 1460, 2127223, 2128682, 2130141, 4528179181405, 4531280671081,
  4534390675481, 205184508064939437814461611, 2053251416575881662428296 1,
  2054659681631676729626856 1,
  4215847321157672995355585100318303042788459718258721 ]
#I  Intermediate result : [ [ 4531280671081, 2054659681631676729626856 1 ],
  [ 1458, 1460, 2127223, 2128682, 2130141, 4528179181405, 4534390675481,
      205184508064939437814461611, 2053251416575881662428296 1,
      4215847321157672995355585100318303042788459718258721 ] ]
```

```
#I
#I  Factors already found : [ 4531280671081, 20546596816316767296268561 ]
#I
#I  Trial division by all primes p < 1000
#I  Intermediate result : [ [ 2, 3, 3, 3, 3, 3, 3 ], [  ] ]
#I  Intermediate result : [ [ 2, 2, 5, 73 ], [  ] ]
#I  Intermediate result : [ [ 7, 303889 ], [  ] ]
#I  Intermediate result : [ [ 2, 1064341 ], [  ] ]
#I  Intermediate result : [ [ 3, 13, 193, 283 ], [  ] ]
#I  Intermediate result : [ [ 5, 11, 521, 158024051 ], [  ] ]
#I  Intermediate result : [ [ 31, 146270666951 ], [  ] ]
#I  Intermediate result : [ [ 61 ], [ 33636804600809743904010 ] ]
#I
#I  Factors already found : [ 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 5, 5, 7, 11,
   13, 31, 61, 73, 193, 283, 521, 303889, 1064341, 158024051, 146270666951,
   4531280671081, 20546596816316767296268561 ]
#I
#I  Trial division by some already known primes
#I
#I  Check for perfect powers
#I
#I  Pollard's Rho
Steps = 16384, Cluster = 1638
Number to be factored :
33636804600809743904010
#I  Intermediate result : [ [ 2251, 149430495783250750351 ], [  ] ]
#I
#I  Pollard's Rho
Steps = 16384, Cluster = 1638
Number to be factored :
205325141657588166624282961
#I
#I  Pollard's Rho
Steps = 16384, Cluster = 1638
Number to be factored :
42158473211576729953555851003183034278845971825872
#I
#I  Factors already found : [ 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 5, 5, 7, 11,
   13, 31, 61, 73, 193, 283, 521, 2251, 303889, 1064341, 158024051,
   146270666951, 4531280671081, 149430495783250750351,
   20546596816316767296268561 ]
#I
#I  Pollard's p - 1
Limit1 = 10000, Limit2 = 400000
Number to be factored :
205325141657588166624282961
#I  Initializing prime differences list, PrimeDiffLimit = 1000000
#I  p-1 for n = 205325141657588166624282961
a : 2, Limit1 : 10000, Limit2 : 400000
#I  First stage
#I  Second stage
#I
```

```
#I  Pollard's p - 1
Limit1 = 10000, Limit2 = 400000
Number to be factored :
42158473211576729953555851003183034278845971825721
#I  p-1 for n = 42158473211576729953555851003183034278845971825721
a : 2, Limit1 : 10000, Limit2 : 400000
#I  First stage
#I  Second stage
#I
#I  Williams' p + 1
Residues = 2, Limit1 = 2000, Limit2 = 80000
Number to be factored :
20532514165758816624282961
#I  p+1 for n = 20532514165758816624282961
Residues : 2, Limit1 : 2000, Limit2 : 80000
#I  Residue no. 1
#I  Residue no. 2
#I
#I  Williams' p + 1
Residues = 2, Limit1 = 2000, Limit2 = 80000
Number to be factored :
42158473211576729953555851003183034278845971825721
#I  p+1 for n = 42158473211576729953555851003183034278845971825721
Residues : 2, Limit1 : 2000, Limit2 : 80000
#I  Residue no. 1
#I  Residue no. 2
#I
#I  Elliptic Curves Method (ECM)
Curves = <func.>
Init. Limit1 = <func.>, Init. Limit2 = <func.>, Delta = <func.>
Number to be factored :
20532514165758816624282961
#I  ECM for n = 20532514165758816624282961
#I  Digits : 26, Curves : 3
#I  Initial Limit1 : 200, Initial Limit2 : 20000, Delta : 200
#I  Curve no.      1 (     3), Limit1 :    200, Limit2 :    20000
#I  Curve no.      2 (     3), Limit1 :    400, Limit2 :    40000
#I  11-digit factor 12077430061 was found in second stage
#I  Intermediate result : [ [ 12077430061, 1700073116718901 ], [  ] ]
#I
#I  Elliptic Curves Method (ECM)
Curves = <func.>
Init. Limit1 = <func.>, Init. Limit2 = <func.>, Delta = <func.>
Number to be factored :
42158473211576729953555851003183034278845971825721
#I  ECM for n = 42158473211576729953555851003183034278845971825721
#I  Digits : 51, Curves : 27
#I  Initial Limit1 : 200, Initial Limit2 : 20000, Delta : 200
#I  Curve no.      1 (    27), Limit1 :    200, Limit2 :    20000
#I  Curve no.      2 (    27), Limit1 :    400, Limit2 :    40000
#I  Curve no.      3 (    27), Limit1 :    600, Limit2 :    60000
                          [ ... ]
```

```
#I   Curve no.      25 (     27), Limit1 :     5000, Limit2 :     500000
#I   Curve no.      26 (     27), Limit1 :     5200, Limit2 :     520000
#I   Curve no.      27 (     27), Limit1 :     5400, Limit2 :     540000
#I
#I   Factors already found : [ 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 5, 5, 7, 11,
   13, 31, 61, 73, 193, 283, 521, 2251, 303889, 1064341, 158024051,
   12077430061, 146270666951, 4531280671081, 1700073116718901,
   149430495783250750351, 2054659681631676729626268561 ]
#I
#I   Multiple Polynomial Quadratic Sieve (MPQS)
Number to be factored :
421584732115767299535558510031830342788459718258721
#I   MPQS for n = 421584732115767299535558510031830342788459718258721
#I   Digits                  :         51
#I   Multiplier              :          1
#I   Size of factor base     :       1326
#I   Prime powers to be sieved  :     1349
#I   Length of sieving interval :   131072
#I   Small prime limit       :        117
#I   Large prime limit       :    3500186
#I   Number of used a-factors :          4
#I   Size of a-factors pool  :         52
#I   Initialization time     :         11 sec.
#I
#I   Sieving
#I
#I   Complete factorizations over the factor base   :      50
#I   Relations with a large prime factor            :       5
#I   Relations remaining to be found                :    1343
#I   Total factorizations with a large prime factor :     614
#I   Used polynomials                               :      67
#I   Elapsed runtime                                :      39 sec.
#I   Progress (relations)                           :       3 %
#I
                            [ ... ]
#I
#I   Complete factorizations over the factor base   :     651
#I   Relations with a large prime factor            :     826
#I   Relations remaining to be found                :       0
#I   Total factorizations with a large prime factor :    9164
#I   Used polynomials                               :    1021
#I   Elapsed runtime                                :     455 sec.
#I   Progress (relations)                           :     100 %
#I
#I   Creating the exponent matrix
#I   Doing Gaussian Elimination, #rows = 1477, #columns = 1378
#I   Processing the zero rows
#I   Dependency no. 1 yielded no factor
                 [ ... ]
#I   Dependency no. 8 yielded no factor
#I   Dependency no. 9 yielded factor 26725378467920336641
#I   The factors are
```

```
[ 26725378467920336641, 15774696422795817479975816558881 ]
#I  Digit partition : [ 20, 32 ]
#I  MPQS runtime : 473.507 sec.

#I  Intermediate result :
[ [ 26725378467920336641, 15774696422795817479975816558881 ], [  ] ]
#I
#I  The result is
[ [ 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 5, 5, 7, 11, 13, 31, 61, 73, 193, 283,
      521, 2251, 303889, 1064341, 158024051, 12077430061, 146270666951,
      4531280671081, 1700073116718901, 26725378467920336641,
      149430495783250750351, 20546596816316767296268561,
      15774696422795817479975816558881 ], [  ] ]

#I  The total runtime was 799.408 sec

[ 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 5, 5, 7, 11, 13, 31, 61, 73, 193, 283,
  521, 2251, 303889, 1064341, 158024051, 12077430061, 146270666951,
  4531280671081, 1700073116718901, 26725378467920336641,
  149430495783250750351, 20546596816316767296268561,
  15774696422795817479975816558881 ]
```

5   How much Time does a Factorization take ?

5.1 The General Factorization Routine

A few words in advance: in general, it is not possible to give a precise prediction for the CPU time needed for factoring a given integer. This time depends heavily on the sizes of the factors of the given number and some other properties which could not be tested before actually doing the factorization. But nevertheless, it is possible to give rough runtime estimates for numbers with given "digit partition" (this means: factors of given sizes).

By default, after casting out the small and other "easy" factors (which should not take more than at most a few minutes for numbers of "reasonable" size) the general factorization routine (see 2.1.1, 2.2.1) uses first ECM (see 3.3.1) for finding factors very roughly up to the third root of the remaining composite and then the MPQS (see 3.5.1) for doing the "rest" of the work (which will often be the most time-consuming part).

Below I will give some timings for ECM and for the MPQS because these methods are by far the most important ones in respect to runtime statistics (the $p \pm 1$-methods (see 3.1.1, 3.2.1) are only suitable for finding factors with certain special properties and CFRAC (see 3.4.1) is just a slower predecessor of the MPQS). All absolute timings are given for a Pentium 200 under Windows.

Logfiles for some examples together with their approximate runtimes can be found on my homepage

> http://www.cip.mathematik.uni-stuttgart.de/~kohlsn/FactInt/Examples.htm

5.2 Timings for ECM

The runtime of `FactorsECM` depends mainly on the size of the factors of the input number. On average, finding a 12-digit factor of a 100-digit number takes about 1 min 40 s, finding a 15-digit factor of a 100-digit number takes about 10 min and finding an 18-digit factor of a 100-digit number takes about 50 min. As a general rule of thumb: one digit more increases the run-time by somewhat less than a factor of two. These timings are very rough, and they may vary by a factor of 10 or more (You can compare trying an elliptic curve with throwing a couple of dice, where a success corresponds to the case where all of them show the same side – it is possible to be successful with the first trial, but it is also possible that this takes much longer. In particular, all trials are independent of one another). In general, ECM is superior to Pollard's Rho for finding factors with at least 10 decimal digits and in the same time needed by Pollard's Rho for finding a 13-digit factor one can reasonably expect to find a 17-digit factor when using ECM, for which Pollard's Rho would need about 100 times as long as ECM. For larger factors this difference grows rapidly. From theoretical calculations it can be stated that finding a 20-digit factor requires about 500 times as much work as finding a 10-digit factor, finding a 30-digit factor requires about 160 times as much work as finding a 20-digit factor and finding a 40-digit factor requires about 80 times as much work as finding a 30-digit factor.

The default parameters are optimized for finding factors with about $15 - 35$ digits. This seems to be a sensible choice since this is the most important range for the application of ECM and for factors having some digits more or less this should be also not too bad. (finding a factor with 30 digits or even more requires a lot of patience – more than I had until now (a 27-digit factor is the largest one I have found so far). ECM usually gives up when the input number n has two factors where both of them are larger

than the third root of n (this is certainly only a rough "probabilistic" statement; in general, sometimes – but seldom – the remaining composite has 3 factors, 4 factors should occur (hardly) never). Certainly, the user could specify other parameters than the default ones – but giving timings for all possible cases here is obviously impossible. The interested reader should follow the references given in the bibliography at the end of this manual for getting information on how much curves with which parameters are usually needed for finding factors of a given size. This depends mainly on the distribution of primes, resp. numbers with prime factors not exceeding a certain bound. About the time needed for trying a single curve with given smoothness bounds for a number of given size (which gives probably the best estimate for precise runtime comparisons with other implementations) I want to mention a typical example: one curve with $(Limit1, Limit2) = (100000, 10000000)$ applied to a 100-digit integer requires a total of 10 min 20 s, where 6 min 45 s are spent for the first stage and 3 min 35 s are spent for the second stage. The time needed for the first stage is approximately linear in $Limit1$ and the time needed for the second stage is somewhat less than linear in $Limit2$.

5.3 Timings for the MPQS

The runtime of `FactorsMPQS` depends only on the size of the input number, not on the size of its factors. Rough timings are as follows: 90 s for a 40-digit number, 10 min for a 50-digit number, 2 h for a 60-digit number, 20 h for a 70-digit number and 100 h for a 75-digit number (this is the size of the largest number I factored by the MPQS until now). These timings are much more precise than those given for ECM, but they may also vary by a factor of about 2 depending on whether a good factor base could be found without using a large multiplier or not. As a general rule of thumb: 10 digits more gives 10 times as much work. For benchmarking purposes, I give the precise timings for some integers that I have observed: $38! + 1$ (45 digits, good factor base with multiplier 1): 2 min 22 s, $40! - 1$ (48 digits, not so good factor base even with multiplier 7): 8 min 58 s, cofactor of $1093^{33} + 1$ (61 digits, good factor base with multiplier 1): 1 h 12 min.

Bibliography

[Bre89]   David M. Bressoud. *Factorization and Primality Testing*. Springer-Verlag, 1989.

[Bre96]   Richard P. Brent. Factorization of the Tenth and Eleventh Fermat Numbers, 1996.

[Coh93]   Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1993.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "`PermutationCharacter`" comes before "permutation group".